Atravesando las Capas de una Aplicación Empresarial: Demostrador Tecnológico J2EE

RODRIGO DE MIGUEL GONZÁLEZ

GRADO EN INGENIERÍA DEL SOFTWARE
FACULTAD DE INFORMÁTICA
DEPARTAMENTO DE INGENIERÍA DEL SOFTWARE E INTELIGENCIA ARTIFICIAL
UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO FIN DE GRADO

Curso 2017-2018

Director: Antonio Navarro Martín

Madrid, 2018

Para ti Claudia, por hacerme sentir el chico más afortunado del mundo.

AGRADECIMIENTOS

Agradezco enormemente a mi tutor de trabajo fin de grado, Antonio Navarro Martín, toda la ayuda prestada y su cercanía a lo largo de todo el desarrollo de este proyecto.

A mi familia por haber hecho esto posible. En especial a mis padres, Fernando y Judit, por sus esfuerzos, sacrificios e incondicional apoyo y dedicación para que consiga mis metas siempre. Y a mi hermana Jimena por seguir aguantándome después de tantos años.

A mis compañeros de facultad Alejandro, Álvaro, Andrea, Beatriz, Carmen, Eduardo, Fernando, Iván, Jefferson, Juan Luis, Laura, Raquel, Raúl, Samuel, Sergio por tantos momentos juntos. Y a Tomás, por su amistad y ayuda.

A ti Claudia por tu amor, tu tiempo y tu ayuda en todo momento.

Y por último, a todos aquellos que no creyeron en mí porque me hicieron más fuerte.

RESUMEN

La arquitectura multicapa es básica en el desarrollo de aplicaciones profesionales. Al estar constituida sobre un conjunto de patrones bien definido, con independencia de la plataforma de desarrollo utilizada, existen diversos marcos de desarrollo centrados en cada una de las capas programables (presentación, negocio e integración).

El objetivo del presente Trabajo de Final de Grado (en adelante TFG) consiste en desarrollar una aplicación web con arquitectura multicapa orientada a servicios, aplicando gran parte de sus patrones. En particular, su aplicación a una determinada plataforma, J2EE en este caso, constituye un demostrador tecnológico de un gran número de marcos de desarrollo de la plataforma J2EE y de herramientas de desarrollo asociadas.

La finalidad es complementar la formación de graduado en Ingeniería del Software con un perfil de desarrollo *Full-Stack* (conocimientos en desarrollo *Front-End*, *Back-End* y servidor). Esto es algo cada vez más demandado por las empresas del sector. Otro objetivo es que este TFG pueda servir de demostrador tecnológico, ilustrando las capacidades de diseño y desarrollo de su creador.

Cabe destacar que la mayoría de las tecnologías utilizadas en este TFG no son estudiadas a lo largo del Grado, por lo que el trabajo no solo ha sido de desarrollo, sino que ha implicado un considerable esfuerzo para investigar el uso y aplicación de cada una de estas tecnologías y herramientas y de su integración en un único proyecto.

ABSTRACT

Multilayer architecture is basic in the development of professional applications. Being constituted on a set of well-defined patterns, regardless of the development platform used, there are several development frameworks focused on each of the programmable layers (presentation, business and integration).

The objective of this Final Degree Project (hereinafter TFG) is to develop a web application with multilayer architecture oriented to services, applying a large part of its patterns. In particular, its application to a specific platform, J2EE in this case, constitutes a technological demonstrator of a large number of J2EE platform development frameworks and associated development tools.

The purpose is to complement the graduate training in Software Engineering with a Full-Stack development profile (knowledge in Front-End, Back-End and server development). This is something increasingly demanded by companies in the sector. Another objective is that this TFG can serve as a technological demonstrator, illustrating the design and development capabilities of its creator.

It should be noted that most of the technologies used in this TFG are not studied throughout the degree, so the work has not only been development, but has involved considerable effort to investigate the use and application of each of them. these technologies and tools and their integration into a single project.

PALABRAS CLAVE

Arquitectura multicapa, *Continuous Integration*, JAAS, JAX-RS, JAX-WS, JPA, JSF, Patrones de Diseño, SOA, *Service Oriented Architecture*

KEY WORDS

Multilayer Architecture, Continuous Integration, JAAS, JAX-RS, JAX-WS, JPA, JSF, Design Patterns, SOA, Service Oriented Architecture

LISTA DE ACRÓNIMOS

API: *Application Programming Interface*
BD: Base de datos
FDI: Facultad De Informática
GPL: *General Public License*
HTML: *HyperText Markup Language*
HTTP: *Hypertext Transfer Protocol*
HTTPS: *Hypertext Transport Protocol Secure*
IDE: *Integrated Development Environment*
J2EE: *Java Platform Enterprise Edition*
JAAS: *Java Authentication and Authorization Service*
JAX-RS: *Java API for RESTful Web Services*
JAX-WS: *Java API for XML Web Services*
JPA: *Java Persistence API*
JPQL: *Java Persistence Query Language*
JSF: *JavaServer Faces*
JSON: *JavaScript Object Notation*
ORM: *Object-Relational Mapping*
POJO: *Plain Old Java Object*
REST: *Representational State Transfer*
SGBDR: Sistema Gestor de Bases de Datos Relacionales
SMTP: *Simple Mail Transfer Protocol*
SOA: *Service Oriented Architecture*
SOAP: *Simple Object Access Protocol*
SQL: *Structured Query Language*
TFG: Trabajo Final de Grado
UCM: Universidad Complutense de Madrid
URL: *Uniform Resource Locator*
WSB: *Web Service Broker*
XHTML: *eXtensible HyperText Markup Language*
XML: *eXtensible Markup Language*

Índice

-

Capítulo 1. Introducción/Introduction

Versión española

El Grado en Ingeniería del Software que imparte la Facultad de Informática de la Universidad Complutense de Madrid proporciona una buena formación al alumno que desee dedicarse al desarrollo de aplicaciones software empresariales. Sin embargo, hay algunos aspectos que no quedan cubiertos por el grado. En particular:

- El desarrollo de una aplicación empresarial con una capa de presentación web.
- El uso de marcos de autorización y autenticación para el acceso a aplicaciones web.
- El uso de una arquitectura orientada a servicios para publicar los servicios de aplicación.
- El uso de conexiones HTTPS para el acceso seguro a servicios web.
- El uso de una herramienta de integración continua que facilite el desarrollo, prueba e integración del software.

Aunque en principio, con la formación recibida, el graduado en Ingeniería del Software podría en una empresa, con un esfuerzo moderado, completar su formación en estos aspectos, el realizar un trabajo fin de grado (TFG) que los cubriese, pondría al egresado en una situación de ventaja con respecto a sus competidores en el mercado laboral.

Por tanto, este trabajo se plantea como una culminación natural del Grado en Ingeniería del Software (GIS), recopilando los conocimientos de programación, bases de datos e ingeniería del software adquirido por los alumnos, y extendiéndolos para aplicarlos en el desarrollo de aplicaciones empresariales.

La arquitectura base para el trabajo es la arquitectura multicapa (Alur et al. 2003; Fowler 2002) que es el fundamento de distintas plataformas de desarrollo empresarial como Oracle J2EE (Alur et al. 2003; Fowler 2002) -Eclipse Jakarta EE en la actualidad- o Microsoft .NET (Freeman, 2017). En la medida en que lo importante es la arquitectura multicapa y sus patrones relacionados, el uso de una plataforma concreta es una cuestión tecnológica, hasta cierto punto, menor. No obstante, la implementación y despliegue de una aplicación empresarial requiere el dominio de una plataforma y de sus marcos asociados. Al ser Oracle Java (Schildt, 2014) el lenguaje de referencia usado por los alumnos en el GIS, Oracle J2EE parecía la opción más razonable para la implementación realizada en este trabajo.

Así planteado, el trabajo requería la formación del alumno para aplicar diversos marcos dentro de la tecnología J2EE. En particular:

- *JavaServer Faces* (JSF) (Geary & Horstmann, 2010) para la capa de presentación. A pesar de disponer de otros marcos para esta capa como Spring MVC (Amuthan G,

2014), se optó por JSF al ser el marco oficial J2EE. Una muy ligera comparativa entre ambos marcos puede verse en (JSF & Spring MVC, 2018).

- *Java Authentication and Authorization Service* (JAAS) (Oaks, 2001) para la autenticación y autorización de la capa de presentación. Dentro de los patrones de seguridad para aplicaciones empresariales (Steel et al, 2005) hay dos que tienen una especial importancia: el *authentication enforcer* y el *authorization enforcer*. Estos patrones se encargan de autenticar y autorizar el acceso de usuarios a aplicaciones. A pesar de su importancia, estos patrones no se ven en asignaturas centradas en patrones de diseño software como Ingeniería del Software o Modelado del Software. Esto se debe a que las plataformas empresariales suelen tenerlos implementados dentro de sus marcos de seguridad, como es el caso de JAAS en J2EE.

- *Java API for XML Web Services* (JAX-WS) (Hansen, 2007) y *Java API for RESTful Web Services* (JAX-RS) (Burke, 2013) para la publicación de servicios de aplicación como servicios web SOAP (Hansen, 2007) y REST (Hansen, 2007), respectivamente. Como ambos tipos de servicios web son utilizados indistintamente por aplicaciones empresariales, en este trabajo se decidió usar ambos marcos. La seguridad en el acceso se ha relegado al protocolo HTTPS y al contenedor web (Apache Tomcat –Vukotic & Goodwill, 2011– en este caso). Esta es una solución potente, sencilla y de bajo coste de desarrollo. Otra opción hubiera sido utilizar el protocolo WS-Security (Hallam-Baker et al, 2003) para los servicios web SOAP, opción que se ha dejado para trabajo futuro por restricciones de planificación.

- JUnit (JUnit, 2018), el principal marco de pruebas unitarias para Java, usado para realizar todas las baterías de pruebas unitarias de cada método programado en el proyecto, probando diferentes entradas y salidas de cada uno de ellos, ya sea con datos erróneos forzando el fallo, campos vacíos, nulos, etc.

- Jenkins (Ferguson, 2011) es un servidor de código abierto para la automatización de tareas y *pipelines* de proyectos software que proporciona una solución de Integración Continua (CI) y entrega continua. Soporta multitud de complementos instalables a modo de *plugins*, como son complementación con CVSs como Git (Hutten, 2017), notificaciones de resultados por *email*, integración con Maven (Varanasi & Belida, 2014), ejecuciones de trabajos con *pipelines*, etc.

El trabajo también requería del uso de otros marcos y tecnologías ya vistas en el grado, tales como:

- *Java Persistence API* (JPA), que es el mecanismo de mapeado objeto-relacional de la plataforma J2EE. Este marco no sólo facilita la persistencia de los objetos del negocio, sino que además proporciona los mecanismos para la gestión de la concurrencia optimista en un entorno transaccional (Fowler, 2002), como la implementada en este trabajo.
- El uso de un sistema de gestión de bases de datos relacionales para la persistencia de los datos. En particular se ha utilizado MariaDB (Bartholomew, 2015) una evolución del conocido Oracle MySQL (Mclaughlin, 2011).

- El uso de un sistema de control de versiones, GitHub (Bell & Beer, 2014) en este caso.

Finalmente, el trabajo necesitaba del uso de una herramienta de integración continua que facilitase la prueba y despliegue de aplicaciones web. En este caso se decidió utilizar Jenkins (Ferguson, J., 2011) en conjunción con Maven para la gestión de librerías.

Por tanto, este trabajo se puede concebir como un demostrador tecnológico que, atravesando todas las capas de una aplicación empresarial, utiliza los principales marcos de desarrollo de la plataforma J2EE:

- JSF y JAAS para presentación y seguridad.
- JAX-WS y JAX-RS para negocio.
- JPA para integración.

Al ser un TFG desarrollado por un único alumno, el cual tenía que formarse en varios marcos de complejidad considerable, el modelo del dominio se simplificó notablemente. El objetivo era contar con un número significativo de entidades que permitiera representar las relaciones más frecuentes en aplicaciones empresariales, manteniendo la complejidad total del proyecto dentro de unos límites razonables.

Por los motivos anteriores, el modelo de proceso utilizado para desarrollar todo el proyecto ha tenido un fuerte enfoque ágil, basado principalmente en Scrum (Schwaber, & Sutherland, 2017), ya que es una metodología muy usada y bien conocida por el alumno. Además, al no conocer inicialmente el alumno la mayoría de tecnologías a utilizar en el proyecto, no cabía utilizar modelos de proceso que dan un mayor peso al diseño previo a la codificación, como, por ejemplo, el Proceso Unificado de Desarrollo (Jacobson, I., 2000). Así, durante todo el proyecto se ha intercalado la adquisición de conocimientos de una tecnología con el desarrollo del proyecto. Esto ha sido posible porque la arquitectura y patrones a utilizar sí que estaban claros y porque el equipo de desarrollo era unipersonal.

La estructura del TFG es la siguiente. El capítulo 2 describe la arquitectura y tecnologías utilizadas. El capítulo 3 describe los requisitos de la aplicación construida. El capítulo 4 describe el diseño de la capa de presentación y negocio de la aplicación cliente. El capítulo 5 describe el diseño de la capa de negocio de la aplicación servidor. El capítulo 6 se centra en la descripción de los mecanismos de prueba y control de versiones. Finalmente, el capítulo 7 describe las conclusiones y el trabajo futuro.

English version

The Degree in Software Engineering taught at the Universidad Complutense de Madrid Computer Science School gives a good training to those students that want to work as developers of software enterprise applications. However, some aspects are not covered by the degree:

- The development of enterprise applications with a web presentation tier.
- The use of authentication and authorization frameworks in the web tier.
- The use of a Service Oriented Architecture in the business logic tier.
- The use of HTTPS connections for securing the access to web services.
- The use of tool for continuous integration that make easier the development, testing and integration of the software.

Although students could use the knowledge achieved in the degree for acquiring required skills in industry, the realization of a degree project in enterprise issues, could lead the student to a favorable position with regard his competitors in the job market.

Therefore, this work is outlined as a culmination of the Degree in Software Engineering, gathering the knowledge in programming, databases and software engineering achieved by the student, and extending them in order to be applied in the development of enterprise applications.

The base architecture in the work is the multitier architecture (Alure et al. 2003; Fowler, 2002), that is the basis for several enterprise platforms such as Oracle J2EE (Alur et al., 2003, Fowler, 2002) – Eclipse Jakarta at present- or Microsoft .NET (Freeman, 2017). Because multitier architecture and its related patterns are the key issue, the use of a concrete development platform is, to some extent, a secondary issue. However, the development and deploy of an enterprise application requires the knowledge of a concrete platform and their associated frameworks. Because Oracle Java (Schildt, 2014) is the main language used by the students in the degree, it seems to be the most reasonable choice for writing the code developed in this work.

Therefore, this work required the student to learn different J2EE frameworks:

- JavaServer Faces, JSF (Geary & Horstmann, 2010) for the presentation tier. Although there are other key frameworks for this tier, such as Spring MVC (Amuthan, 2014), we choose JSF because is the official J2EE framework. There is a brief comparison between these frameworks in (JSF & Spring MVC, 2018).
- Java Authentication an Authorization Service (JAAS) (Oaks, 2001) for the authentication and authorization in the presentation tier. Steel et al. (2005) identify authentication and authorization enforcer as two key patterns. These patterns are not taught in the degree, because most enterprise applications include them in their frameworks, as is the case of JAAS in J2EE.

- Java API for XML Web Services (JAX-WS) (Hansen, 2007) and Java API for RESTful Web Services (JAX-RS) (Burke, 2013) for SOAP (Hansen, 2007) and REST (2007) publication of web services. Because both types of services are used in enterprise applications, they were included in this work. The secure access to these services is delegated in the web container (Apache Tomcat –Vukotic & Goodwill, 2011– in this case). This is a simple, powerful and low-cost solution. Other solution is the use of the WS-Security protocol (Hallma-Baker et al., 2003) for the SOAP web services. This solution is considered for future work.

- JUnit (JUnit, 2018), the main framework for software testing, used for testing the business logic in the project.

- Jenkins (Ferguson, 2011) an open code server for task automation and pipelines in software projects that supports continuous development and integration. It can include several CVS plugins such as Git (Hutten, 2017), email notifications, Maven integration (Varanasi & Belida, 2014), pipelines execution, etc.

This work also required the use other frameworks and technologies taught in the degree, such as:

- Java Persistence API (JPA) the J2EE object-relationship framework. This framework provides support for persistence of domain objects, dynamic loading, transactional management and optimistic management of the concurrency (Fowler, 2002) as the one implemented in this work.
- The use of a relational database management system for data persistence, MariaDB (Bartholomew, 2015) in this work, an evolution of Oracle MySQL (Mclaughlin, 2011).
- The use of a CVS, GitHub (Bell & Beer, 2014) in this work.

Therefore, this work can be considered as a technological demonstrator, that, going through all the layers of an enterprise application, it uses the main development frameworks of J2EE platform:

- JSF and JAAS for presentation and security.
- JAX-WS and JAX-RS for business.
- JPA for integration.

Because this work is made by a single student that had to learn several complex frameworks, the domain model was simplified. The goal is to have enough entities to show the most common relationships in enterprise applications, keeping the complexity of the project under some reasonable limits.

Due to these restrictions, the process model used in this project, has had an agile approach, based on Scrum (Schwaber, & Sutherland, 2017) a known process by the student. Due to the lack of knowledge of the development frameworks, it was not possible to use model processes based on designing before programming, such as the Rational Unified Process (Jacobson, I., 2000). Thus, during the process, learning and programming have been intermixed.

This work is structured in the following chapters. Chapter 2 describes the architecture and technologies used in this project. Chapter 3 describes the software requirements. Chapter 4 describes the design of the presentation of business tiers of the client application. Chapter 5 describes the design of the business tier of the server application. Chapter 6 focusses on testing and continuous integration. Finally, Chapter 7 provides conclusions and future work.

Capítulo 2. Arquitectura, Marcos y Tecnologías Utilizadas

Grosso modo, este proyecto consta de dos aplicaciones desplegadas en Tomcat, una que juega el papel de cliente (el *front-end*) y otra que juega el papel de servidor (el *back-end*) responsable por tanto de la persistencia de datos. Ambas están unidas por servicios web REST y SOAP. En su desarrollo se ha utilizado Jenkins, que hace uso de GitHub, Maven y JUnit para gestionar el código. En este capítulo se describen la arquitectura, marcos y tecnologías utilizadas.

2.1 Arquitectura multicapa

La arquitectura multicapa (Alur, Crupi y Malks, 2003; Fowler, 2002) es una de las más utilizadas en el desarrollo de aplicaciones empresariales. Consiste en dividir el código en tres capas independientes, pero interconectadas, favoreciendo así el objetivo de conseguir dividir el sistema en componentes altamente cohesivos y débilmente acoplados (Booch, 96).

La arquitectura define cinco capas, tal y como ilustra la Figura 1.

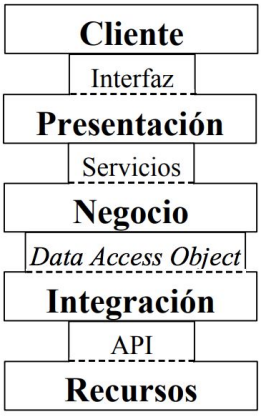

Figura 2.1.1. *La arquitectura multicapa*

- La *capa de cliente*, representa a los clientes que acceden al sistema.

- La *capa de presentación*, encapsula toda la lógica desarrollada para dar soporte a todas las peticiones del cliente y el control de seguridad de acceso al sistema. En este TFG ha sido implementada con JSF (implementado con Mojarra (Oracle JSF, 2018)) y JAAS.

- La *capa de negocio***,** es el núcleo del sistema, contiene toda la lógica de la aplicación que implementa los procesos de negocio. En este proyecto, dicha lógica es expuesta como servicios web REST y SOAP, y ha sido programada con las APIs JAX-WS y JAX-RS implementadas en Apache CXF (Balani & Hathi, 2009).

- La *capa de Integración*, encapsula toda la complejidad del acceso a los recursos persistentes, incluyendo gestión transaccional. En este trabajo ha sido implementada mediante el marco de mapeado objeto relacional (ORM) JPA, implementado con Hibernate (Mihalcea, 2016).

- La *capa de recursos*, es la encargada de contener todos los datos persistentes. Este proyecto tiene tres bases de datos relacionales que residen en el SGBDR MariaDB. Estas tres bases de datos se destinan a persistir: los datos de la aplicación, los

usuarios con roles de acceso a la aplicación cliente y la última, para el control de acceso a través del protocolo HTTPs.

Este proyecto software consta de dos aplicaciones Java distintas, el *front-end* y el *back-end*, tal y como describe el diagrama de componentes de la Figura 2.1.2.

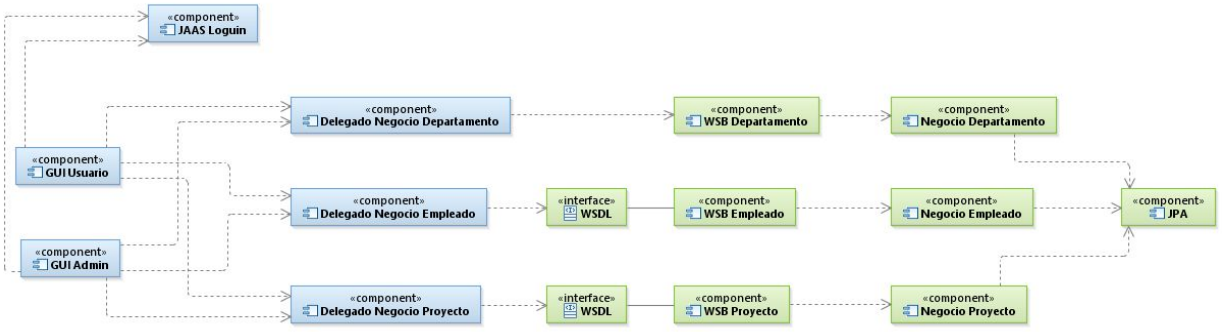

Figura 2.1.2. *Diagrama de Componentes de la aplicación. Los componentes del front-end aparecen en azul, y los del back-end en verde*

El *front-end* es una aplicación web con presentación JSF, que se conecta mediante servicios web SOAP JAX-WS (gestión de empleados y proyectos) y REST JAX-RS (gestión de departamentos) con el *back-end* para la obtención de los recursos solicitados. Con JAAS se restringe el acceso a las páginas web del *front-end* en función del rol que tienen los distintos usuarios que accedan al sistema. En el proyecto solo se han implementado el rol de *Administrador* (con acceso total) y el rol de *Usuario* (con acceso únicamente a las funciones de visualización de datos). Para el almacenamiento de las contraseñas de los usuarios se utiliza una base de datos relacional alojada en MariaDB y accedida mediante JPA.

Los servicios web JAX-WS y JAX-RS han sido implementados utilizando Apache CXF. La seguridad de acceso a estos servicios se ha delegado en el contenedor de los mismos, Apache Tomcat, el cual necesita una base de datos (alojada también en MariaDB) para el control de acceso a través del protocolo HTTPS. Dichos usuarios representan a las distintas aplicaciones cliente que se podrían conectar a servidor.

El proyecto hace uso de múltiples librerías, que han sido gestionadas mediante Maven, que gestiona fácilmente la inyección de dependencias en el proyecto, su estructura, la ejecución de pruebas unitarias mediante JUnit, y la construcción del proyecto.

Para el alojamiento del código se ha hecho uso de SCV GitHub, usado dos repositorios distintos, uno para el *front-end*[1] y otro para el *back-end*[2]. Ambos poseen dos ramas distintas: la *Master* que contiene código finalizado y que pasa las pruebas, y la *Development* que contiene el código que está siendo desarrollado.

[1] https://github.com/hunzaGit/TFG_cliente
[2] https://github.com/hunzaGit/TFG_server

Por otra parte, se ha usado el servidor Jenkins, que proporciona una solución de Integración Continua para el desarrollo de todo el proyecto, siendo usado para la automatización de la construcción, ejecución de pruebas, y despliegue en Apache Tomcat.

El despliegue de ambas aplicaciones web queda plasmado en el diagrama de despliegue de la Figura 2.1.3.

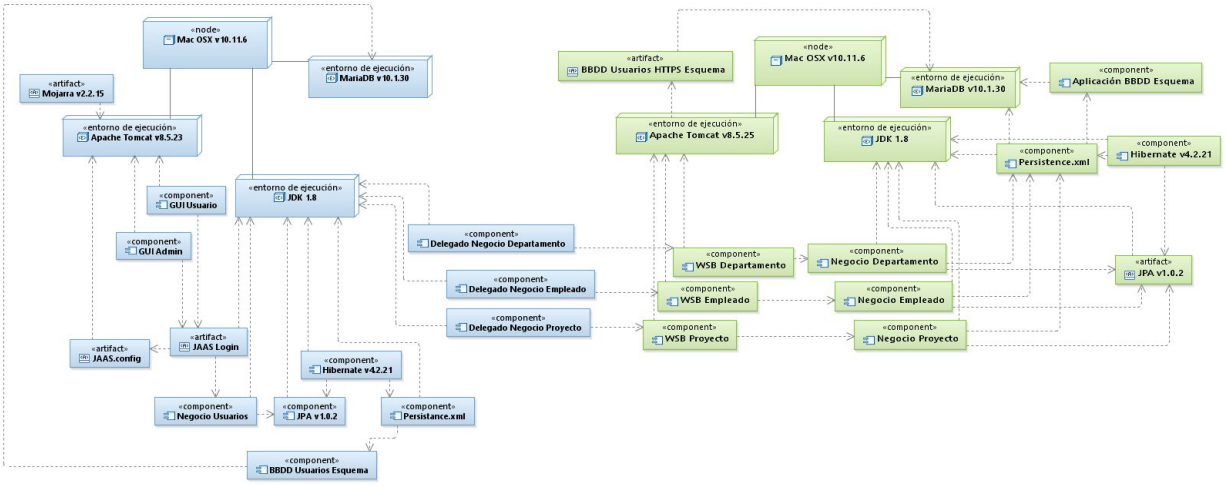

Figura 2.1.3. *Diagrama de despliegue*

La Figura 2.1.4 muestra los marcos y tecnologías utilizados en el proyecto.

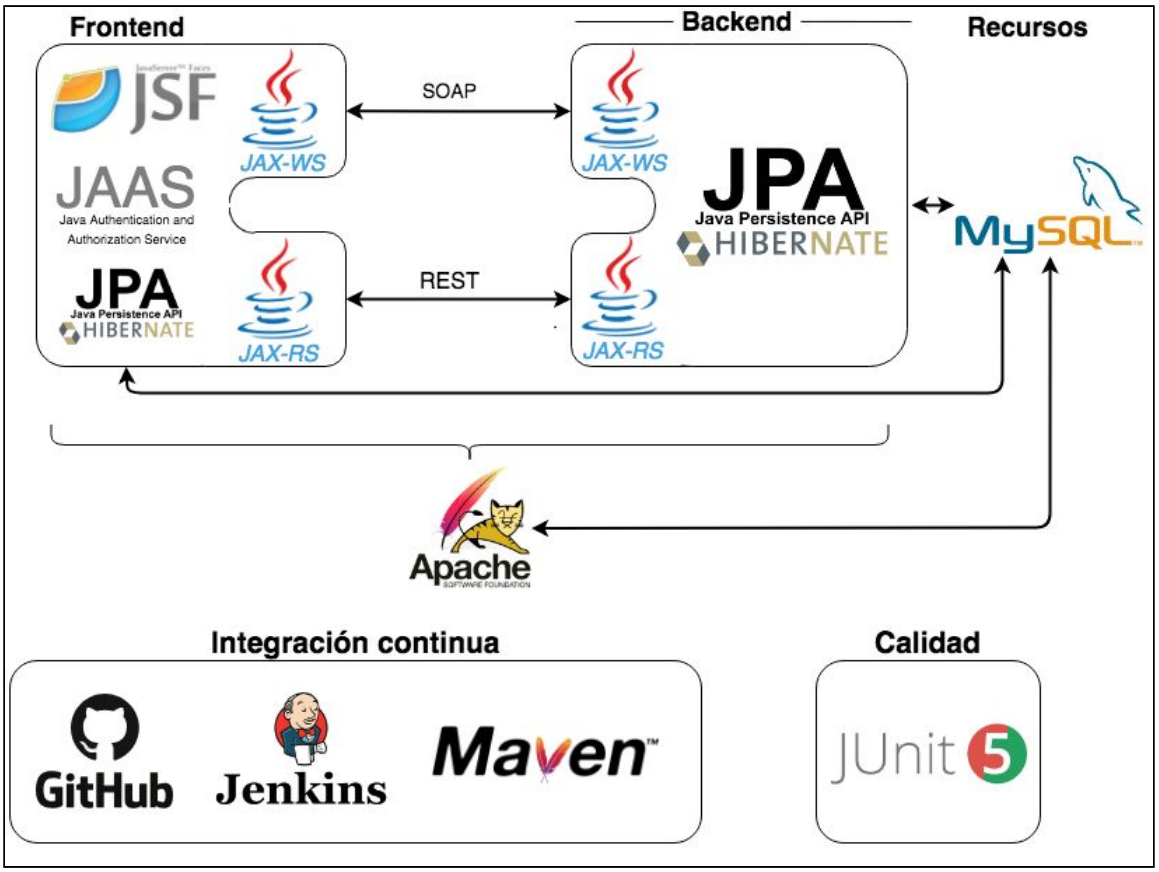

Figura 2.1.4. *Marcos y tecnologías utilizados en el proyecto*

2.2 Tecnologías de desarrollo J2EE

2.2.1 JSF

JavaServer Faces (JSF) es un marco de desarrollo construido sobre el patrón modelo-vista-controlador (Fowler, 2002) para aplicaciones Java J2EE basadas en tecnologías web. JSF utiliza JSP (*JavaServer Pages*) (Zambon, 2012) como tecnología para la gestión de las vistas y el despliegue de páginas web. En el proyecto ha sido usada la versión JSF 2.2.15 y PrimeFaces 6.1 (PrimeTek, 2017).

En JSF, las peticiones HTTP son tratadas por clases Java especiales llamadas *Managed Bean* (Geary & Horstmann, 2010, que gestionan todos los recursos de esa página, y se encargan de invocar la lógica de aplicación (son por tanto una especie de comandos (Gamma et al, 1994)). Posteriormente se genera la vista, mediante plantillas XHTML (Musciano, 2006) -en este proyecto-, obteniendo páginas HTML (Duckett, 2011) que son enviadas al cliente e interpretadas en el navegador.

Los *Managed Bean* son POJOs (*Plain Old Java Objects*) administrados por JSF mediante anotaciones (o ficheros de configuración XML). Contienen como atributos todos los datos (valores de los formularios, o elementos de representación de información) usados en las plantillas JSF, con sus respectivos *getter* y *setter,* y los métodos con el código de las distintas acciones que se pueden ejecutar.

A modo de ejemplo simplificado se muestra a continuación la clase **EmpleadoBean** (el *Managed Bean* del módulo Empleado). Con la primera anotación se convierte la clase en un *Managed Bean* y el nombre con el que se identifica y, con la segunda, se le indica a JSF que el ciclo de vida del *bean* será el de la sesión HTTP. Como los *getter* y *setter* coinciden en nombre con el atributo no hace falta usar anotación para identificarlo.

```
@ManagedBean(name = "EmpleadoBean")
@SessionScoped
public class EmpleadoBean implements Serializable {

	private String email;

	public String getEmail() { return email; }

	public void setEmail(String email) { this.email = email; }

}
```

Figura 2.2.1. Ejemplo de clase EmpleadoBean

Respecto a la plantilla JSF, usando la etiqueta `<h:outputText>` se puede enlazar con el atributo del bean:

```
<strong>Email:</strong>
<h:outputText value="#{EmpleadoAction.empleadoCompleto.empleado.email}"/>
```

Figura 2.2.2. Enlace de HTML con atributo de EmpleadoBean

Por otra parte, JSF permite declarar las reglas de navegación de todo el conjunto de páginas de la aplicación. Estas reglas se definen en el fichero *`faces-config.xml`*. En la plantilla JSF de, por ejemplo, la página de inicio haciendo uso de la etiqueta `<h:commandLink>` con los valores `value` y `action` se puede enlazar esta vista con la de administración de forma que, al seleccionar el enlace HTML, JSF cargará la vista del fichero `/views/admin/admin.xhtml`.

Así, la Figura 2.2.3 describe en enlace incluido en `index.xhtml`.

```
<h:commandLink value="Admin" action="a_admin"/><br/>
```

Figura 2.2.3. Enlace XHTML para navegar entre vistas con reglas de navegación

La Figura 2.2.4 describe la regla de navegación definida en el fichero `faces-config.xml`.

```
<navigation-rule>
      <from-view-id>/views/index.xhtml</from-view-id>
      <navigation-case>
          <from-outcome>a_admin</from-outcome>
          <to-view-id>/views/admin/admin.xhtml</to-view-id>
          <redirect/>
      </navigation-case>
  </navigation-rule>
```

Figura 2.2.4. Regla de navegación para el ejemplo anterior

2.2.2 Seguridad

2.2.2.1 JAAS

Java Authentication and Authorization Service (JAAS) es la API que provee Java para la autorización y control de acceso de los usuarios al sistema desde las vistas mediante un sistema de autenticación y autorización basado en roles. Integrado en la máquina virtual de Java desde la versión 1.4, tiene como objetivo simplificar y abstraer al programador de casi todo el proceso. En el proyecto ha sido usada la versión incluida en el JDK 1.8.
JAAS se utiliza en el proyecto para proveer de seguridad los accesos a las páginas web de la aplicación cliente del proyecto.

Para poder usar JAAS en un proyecto hay que realizar dos configuraciones distintas: una en el código de la aplicación y otra en la configuración de Apache Tomcat mediante la declaración de un JAASRealm.

La aplicación requiere de varias clases JAAS: la clase `UserPrincipal` representa al usuario que, con un login basico (usuario y contraseña), accede al sistema; la clase `RolePrincipal` representa el rol del usuario persistido en la base de datos; la clase `LoginModule` es donde JAAS delega el control para realizar la comprobación de que los datos son correctos y autorizar o no el acceso al recurso solicitado en la vista.

Tal y como describe la figura 2.2.5. en el fichero *`web.xml`* se agregan los distintos roles que tienen los usuarios (guardados en la base de datos con el usuario y la contraseña) y las restricciones de seguridad de cada recurso.

```
<!-- ********* ROLES ********* -->
<security-role>
   <description>Administrador</description>
   <role-name>ADMIN</role-name>
</security-role>

<!-- ********* RESTRICCIONES DE RECURSOS ********* -->
<security-constraint>
   <web-resource-collection>
       <web-resource-name>Admin</web-resource-name>
       <url-pattern>/views/Modulo_Administrador/*</url-pattern>
   </web-resource-collection>

   <auth-constraint>
       <role-name>ADMIN</role-name>
   </auth-constraint>
</security-constraint>
```

Figura 2.2.5. Declaración de rol y restricción de acceso asociada

Por otro lado, en la configuración de Apache Tomcat hay que agregar al fichero *$CATALINA_BASE/conf/jaas.config* la clase del proyecto encargada de realizar el login, tal y como describe la Figura 2.2.6.

```
TFG_cliente {
    com.rodrigo.TFG_cliente.Presentacion.seguridad.jaas.LoginModule
required debug=true;
};
```

Figura 2.2.6. *Configuración en Apache Tomcat del LoginModule*

En la sección "Execute The Requested Command" del fichero *$CATALINA_BASE/bin/catalina.sh* hay que agregar una directiva de *JAVA_OPTS* (son una serie de opciones que se ejecutan antes de lanzar el servidor con la aplicación) para indicar a Tomcat que la configuración de JAAS y el login están en ese fichero, tal y como describe la Figura 2.2.7.

```
JAVA_OPTS="$JAVA_OPTS
-Djava.security.auth.login.config=$CATALINA_BASE/conf/jaas.config"
```

Figura 2.2.7. *Configuración en `catalina.sh` para arrancar JAAS*

Con estos pasos, cuando Apache Tomcat despliegue la aplicación sabrá que tiene un *Realm* asignado, y cuando un usuario solicite un recurso restringido redirigirá a la vista de login, y una vez enviado, cargará la clase `LoginModule` para la autorización del usuario, que si su rol se lo permite, le redirigirá automáticamente al recurso solicitado

2.2.2.2 Seguridad en Apache Tomcat y servicios web con JAX-WS y JAX-RS

Hypertext Transport Protocol Secure es el protocolo para la transferencia segura de datos basado en HTTP. Los mensajes son cifrados usando SSL/TLS para crear un canal seguro de punto a punto para el envío de información sensible, de esta forma se evita que un atacante que haya interceptado la transferencia pueda leer la información del mensaje.

HTTPS se utiliza en el proyecto para asegurar la conexión entre la aplicación cliente y la implementación de los servicios web en la aplicación servidor. Así, usando los marcos JAX-WS y JAX-RS más la configuración de Apache Tomcat, se ha implementado el uso de HTTPS en el envío de mensaje a los servicios web expuestos.

Para poder llevar a cabo esto hay que crear un *Keystore* (un almacén de claves) en la que alojar los certificados que usarán ambas aplicaciones.

En `$CATALINA_BASE/seguridad` se ejecuta el siguiente comando:

“`keytool -genkey -alias tomcat -keyalg RSA -keystore ".keystore"`”

Configuran con los siguientes datos:

“`CN=localhost, OU=localhost, O=TFG, L=Madrid, ST=Madrid, C=ES`”.

Posteriormente se exporta el certificado en formato `x.509` usando el comando:

```
keytool -export -alias tomcat -keystore ".keystore" -rfc
-file tomcat.cer
```

En el directorio `$JDK_HOME/Contents/Home` se importa el certificado creado anteriormente a la máquina virtual de Java con el comando:

```
sudo keytool -import -trustcacerts -keystore
jre/lib/security/cacerts -storepass changeit -noprompt
-alias tomcat -file $CATALINA_BASE/seguridad/tomcat.cer
```

Una vez realizados estos pasos, se ha de configurar el servidor Apache Tomcat (fichero `server.xml`) para que redireccione al puerto 8443 (utilizado por convenio para el protocolo HTTPS) las peticiones que entren por los puertos 8080 y el 8009. Se le indica la ruta del certificado creado anteriormente y se le configura un *Realm* con los datos de la BBDD de usuarios y roles con permiso para usar el HTTPS.

Dentro del `<Service name="Catalina">` se configura el conector de *endpoints* con las etiquetas de la figura 2.2.8.

```
<Connector connectionTimeout="20000" port="8080" protocol="HTTP/1.1"
redirectPort="8443"/>
     ……
<Connector port="8009" protocol="AJP/1.3" redirectPort="8443"/>
```

Figura 2.2.8. *Configuración en `server.xml` para redireccionar las peticiones a HTTPS*

Ahora se indica la ruta del certificado para su uso como indica la Figura 2.2.9:

```
<Connector SSLEnabled="true" maxThreads="150" port="8443"
protocol="org.apache.coyote.http11.Http11NioProtocol">
    <SSLHostConfig>

        <Certificate
            certificateKeystoreFile="$CATALINA_BASE/seguridad/.keystore"
            type="RSA" />

    </SSLHostConfig>
</Connector>
```

Figura 2.2.9. *Configuración en `server.xml` para usar el certificado.*

Antes del siguiente paso en el servidor Apache Tomcat, hay que crear una base de datos nueva que usará Tomcat para almacenar los usuarios que pueden usar los servicios web de la aplicación servidor bajo protocolo HTTPS (no confundir con los usuarios de Apache Tomcat del fichero `tomcat-users.xml`).

Esta base de datos contiene dos tablas, una para los usuarios y otra para los roles. La de usuario contiene el nombre del usuario y la contraseña y la de roles el nombre de usuario y el rol que tiene.

Por último, en el *`server.xml`*, se crea un Realm con todos los datos de acceso a la BBDD, indicando las dos tablas y el nombre de las columnas, con los usuarios, contraseñas y los roles de acceso, como se puede ver en la figura 2.2.10.

```
<Realm
    className="org.apache.catalina.realm.JDBCRealm"
    connectionURL="jdbc:mysql://localhost:3306/TFG_BBDD_tomcat?
        serverTimezone=UTC&amp;user=root&amp;password="
    driverName="com.mysql.cj.jdbc.Driver"
    userRoleTable="user_roles"
    userTable="users"
    roleNameCol="role_name"
    userCredCol="user_pass"
    userNameCol="user_name"/>
```

Figura 2.2.10. *Configuración en `server.xml` para el acceso de Apache Tomcat a la BBDD*

Una vez configurado Apache Tomcat hay que configurar las aplicaciones cliente y servidor para el correcto uso del protocolo HTTPS.

En este proyecto se han creado dos usuarios (con dos roles distintos) para SOAP y REST con el fin de simular dos aplicaciones distintas, aunque también se podría haber usado sólo uno. El proceso de configuración es el mismo para ambos. A continuación, se explica cómo hacerlo en el fichero `web.xml` de la aplicación servidor.

La Figura 2.2.11 describe como crear un rol.

```
<security-role>
    <description>usuario escritura</description>
    <role-name>escritura</role-name>
</security-role>
```

Figura 2.2.11. *Configuración de un rol de acceso a la aplicación en el `web.xml`*

Después se crea una nueva restricción de seguridad, en la que se indica el recurso que se quiere proteger, los roles que tiene acceso a dicho recurso y la restricción de indica la fuerza de la protección que se usará. En la Figura 2.2.12 es `CONFIDENTIAL`, que garantiza un transporte confidencial de los datos. `INTEGRAL` serviría para garantizar la integridad de los datos y `NONE` para aceptar cualquier petición.

```
<security-constraint>
        <web-resource-collection>
            <web-resource-name>HTTPS usuario escritura</web-resource-name>
            <url-pattern>/services/SA_Empleado</url-pattern>
        </web-resource-collection>

        <auth-constraint>
            <role-name>escritura</role-name>
        </auth-constraint>

        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
```

Figura 2.2.12. *Configuración de una restricción de seguridad en el* `web.xml`

Con esto queda todo configurado para que tanto Apache Tomcat, como la aplicación servidor, funcionen bajo HTTPS y acepten o rechacen las peticiones si las reciben bajo un protocolo no seguro o de un usuario no permitido.

Desde las aplicaciones cliente solo hará falta indicar el usuario y contraseña para el envío de los mensajes, lo que se describe en las siguientes secciones.

2.2.3 JAX-WS

Java API for XML Web Services (JAX-WS) es una tecnología para la creación de servicios web SOAP mediante el intercambio de mensajes XML. Forma parte de la plataforma J2EE y hace uso de las anotaciones para facilitar el desarrollo y despliegue de los *endpoint*. Para este proyecto se ha utilizado la implementación Apache CXF 3.2.0.

JAX-WS hace uso del protocolo SOAP para el envío de los mensajes, carentes de estado y con la posibilidad de ser usados para definir otros protocolos más complejos. Los servicios SOAP dependen de un documento descriptor, el WSDL (*Web Services description Language*) (Hansen, 2007) en formato XML en el que se detalla la interfaz pública de los servicios web, los requisitos del protocolo usado y el formato de los mensajes.

A modo de ejemplo muy simplificado se muestran las tres partes esenciales para implementar un servicio web con JAX-WS. Para exponer los servicios web se hace uso del patrón *Web Services Broker*, WSB, (Alur et al., 2003).

En la interfaz del servicio (Figura 2.2.8) basta con usar la anotación `@WebService`, y en el método la anotación `@WebMethod` y su nombre.

```
@WebService
public interface IBroker_SA_Empleado {

    @WebMethod(operationName="listarEmpleados")
    public List<TEmpleado> listarEmpleados();
}
```

Figura 2.2.13. Ejemplo de la interfaz Web Service Broker SOAP de Empleado

En la implementación (Figura 2.2.14) se especifica en el servicio la dirección de la interfaz con la jerarquía de paquetes del proyecto y el nombre del servicio.

```
@WebService(
        endpointInterface= "<PAQUETES>.IBroker_SA_Empleado",
        serviceName="Broker_SA_EmpleadoImpl")
public class Broker_SA_EmpleadoImpl implements IBroker_SA_Empleado {

    public List<TEmpleado> listarEmpleados() {
        return FactoriaSA.getInstance()
                  .crearSA_Empleado().listarEmpleados();
    }
}
```

Figura 2.2.14. Implementación del Web Service Broker SOAP de Empleado

En el `cxf-servlet.xml` se expone el servicio web bajo una dirección y especificando quién es la clase que implementa el servicio. La clase `LoggingFeature` se usa para registrar la entrada y salida de los mensajes en el sistema. Ambos aspectos se describen en la Figura 2.2.10.

```
  <jaxws:endpoint id="Broker_SA_Empleado"
                    implementor="<PAQUETES>.Broker_SA_EmpleadoImpl"
                    address="/SA_Empleado">

        <jaxws:features>
            <bean class="org.apache.cxf.feature.LoggingFeature"/>
        </jaxws:features>
  </jaxws:endpoint>
```

Figura 2.2.15. Declaración del endpoint SOAP en `cxf-servlet.xml`

En la parte cliente se tiene la interfaz del servicio (Figura 2.2.11) creada a partir del WSDL expuesto por el servidor con dos cambios importantes en la anotación `@WebService`, `name` es el nombre de la clase que implementa el servicio y `targetNamespace` es la jerarquía de paquetes en sentido inverso.

```
@WebService(
        targetNamespace =
"http://impl.Serv_aplicacion.Modulo_Empleado.Negocio.TFG_server.rodrigo.com/",
        name = "Broker_SA_EmpleadoImpl")
public interface IBroker_SA_Empleado {

    @WebMethod(operationName="listarEmpleados")
    public List<TEmpleado> listarEmpleados();
}
```

Figura 2.2.16. Interfaz del servicio en aplicación cliente

Para la creación y uso del servicio se ha de crear el puerto de conexión. Este proceso es algo complejo, pero el resultado es que se accede a las operaciones que expone el servicio desde el puerto como si fuese un método más de clase, abstrayendo completamente al programador de los entresijos del protocolo.

Los pasos a seguir son seis (numerados en los comentarios de la Figura 2.2.12):

1. Se crea una `QName` con el `targetNamespace` y el `name` de la interfaz del servicio.
2. Se crea la URL desde la que se puede acceder al fichero WSDL del servicio bajo HTTP. En el ejemplo de la figura, es diferente a la URL del propio servicio debido a que éste está bajo HTTPS y solo se puede acceder tras crear el puerto.
3. Se crea un `Service` con la URL del WSDL y la `QName` anterior.
4. Al servicio se le pide un puerto del tipo `IBroker_SA_Empleado.class`. Con esto ya tenemos el puerto listo para usarse, pero aún se ha de configurar el usuario y contraseña para poder acceder a los servicios expuestos bajo HTTPS.
5. Con el puerto y mediante un *cast* a `BindingProvider` se crea un `RequestContext` al que se le agregan el usuario y contraseña.
6. Por último, se le agrega la URL del *endpoint* como una propiedad.

```
private IBroker_SA_Empleado portEmpleados;

public Delegado_EmpleadoImpl() throws ProxyException {

    log.info("1 Creando Qname del servicio");
    QName SERVICE_EMPLEADO = new QName(NAMESPACE_URI, SERVICE_NAME);

    log.info("2 Creando URL_WSDL de enlace");
    URL wsdlURLEmpleados = new
            URL("http://localhost:8080/TFG_server/wsdl/SA_Empleado.wsdl");


    log.info("3 Creando servicio Empleado");
    Service ssEmpleados = Service.create(wsdlURLEmpleados, SERVICE_EMPLEADO);


    log.info("4 Creando puerto de enlace para el servicio");
    portEmpleados =  ssEmpleados.getPort(IBroker_SA_Empleado.class);


    log.info("5 Asignando usuario y contraseña");
    Map<String, Object> req_ctx= ((BindingProvider)
        portEmpleados).getRequestContext();

    req_ctx.put(BindingProvider.USERNAME_PROPERTY, "usuario");
    req_ctx.put(BindingProvider.PASSWORD_PROPERTY, "contraseña");

    log.info("6 Asignación del puerto seguro");
    String ep = "https://localhost:8443/TFG_server/services/SA_Empleado";
    req_ctx.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, ep);

}
```

Figura 2.2.17. Creación del puerto en el Delegado de Empleado

Tras estos pasos el servicio está configurado completamente y listo para ser usado. Como se puede ver en la Figura 2.2.13 el acceso a los métodos del servicio es trivial.

```
public class Delegado_EmpleadoImpl extends Delegado_Empleado {

    private IBroker_SA_Empleado portEmpleados;

    @Override
    public List<TEmpleado> listarEmpleados() {
        return portEmpleados.listarEmpleados();
    }
}
```

Figura 2.2.18. Acceso a un método del puerto del servicio

2.2.4 JAX-RS

Java API for RESTful Web Services (JAX-RS) es una tecnología para la creación de servicios web de arquitecturas cliente-servidor mediante la arquitectura REST (Hansen, 2007). Forma parte de la plataforma J2EE y hace uso de las anotaciones para facilitar el desarrollo y despliegue de los *endpoint*. Para este proyecto se ha utilizado la implementación Apache CXF 3.2.0.

REST utiliza directamente el protocolo HTTP y HTTPS y sus verbos (GET, POST, PUT y DELETE) para el intercambio de datos en cualquier formato (JSON, XML, etc.). Los mensajes HTTP usados, al ser de un protocolo sin estado, deben contener toda la información necesaria para resolver la petición.

A modo de ejemplo muy simplificado se muestran las dos partes esenciales para implementar un servicio web con JAX-RS. Para exponer los servicios web se hace uso del patrón *Web Service Broker* aunque esta vez no se requiere la interfaz para exponer los *endpoint*.

En la implementación del servicio usando la anotación `@Path` sobre la clase se especifica la dirección HTTP bajo las que se va a exponer y sobre el método se indica la URI, el verbo de la operación y el formato de la respuesta, tal y como describe la Figura 2.2.14.

```
@Path("/departamento")
public class Broker_SA_DepartamentoImpl implements
IBroker_SA_Departamento {

    @GET
    @Path("/{id}")
    @Produces("application/xml")
    @Override
    public Response buscarByID(@PathParam("id") Long id) {

        .........
        TDepartamentoCompleto dept = FactoriaSA
                                        .getInstance()
                                        .crearSA_Departamento()
                                        .buscarByID(id);

        return Response
                  .status(Response.Status.OK)
                  .entity(dept)
                  .build();
    }
}
```

Figura 2.2.19. Implementación del Web Service Broker REST de Departamento

En el *cxf-servlet.xml* se declara un *Bean* con la clase implementadora del servicio y expone bajo una dirección, tal y como describe la Figura 2.2.15.

```
<bean
      id="broker_SA_Depart"
      class="<PAQUETES>.Broker_SA_DepartamentoImpl" />

<jaxrs:server
      id="Broker_Departamento"
      address="/SA_Departamento">

    <jaxrs:serviceBeans>
        <ref bean="broker_SA_Depart" />
    </jaxrs:serviceBeans>
</jaxrs:server>
```

Figura 2.2.20. Declaración del endpoint REST en `cxf-servlet.xml`

Por otra parte, la invocación desde el cliente se realiza de la siguiente forma (a modo muy simplificado): antes de realizar las peticiones hay que crear un cliente con la clase `ClientBuilder`, al que se le pasa como argumento un objeto `Authenticator` (encargado de añadir la cabecera correspondiente a la petición con el usuario y contraseña indicados, Figura 2.2.16) con el usuario y la contraseña para el control de acceso que hará Apache Tomcat al estar funcionando bajo HTTPS. La Figura 2.2.17 describe este proceso.

```
Authorization: Basic base64("user:password")
```

Figura 2.2.21. Cabecera `Authorization` para HTTPS

```
public Delegado_DepartamentoImpl() {

        log.info("Creando cliente");
        Client cliente = ClientBuilder
                .newBuilder()
                .newClient()
                .register(new Authenticator("user", "pass"));

        log.info("DelegadoDelNegocio creado");
    }
```

Figura 2.2.22. Creación del cliente REST para realizar las peticiones HTTPS

Una vez creado el cliente, se puede realizar la invocación segura de los distintos métodos. La Figura 2.2.18 continua con el ejemplo anterior: se indica la URL completa del recurso que se quiere acceder y se añade el id para completar la URI y realizar la petición GET, pasando como argumento la clase del objeto que se va a recibir para que realice la deserialización automáticamente.

```
public TDepartamentoCompleto buscarByID(Long id) throws
DepartamentoException {

    WebTarget wt = cliente.target("<URL>");
    wt = wt.path(id.toString());
    Invocation.Builder b = wt.request();

    TDepartamentoCompleto dept = b.get(TDepartamentoCompleto.class);

    return dept;

}
```

Figura 2.2.23. Invocación de una operación REST mediante el cliente HTTPS

2.2.5 JPA

JPA es el marco ORM (*Object-Relational Mapping*) del estándar J2EE. Facilita enormemente el trabajo de mapear los objetos de la aplicación a las relaciones de una base de datos relacional mediante el uso de anotaciones o ficheros XML de configuración. En el proyecto se ha usado la versión JPA 1.0.2 implementada con Hibernate 4.2.21.

JPA resuelve los cuatro problemas básicos del Almacén del Dominio referidos a los Objetos del Negocio (Fowler et al., 2003):

- La persistencia de los objetos.
- La carga dinámica de los mismos.
- La gestión de transacciones con la base de datos.
- La concurrencia de varios hilos o procesos.

Con anotaciones sobre las clases, se indican propiedades para el mapeo de las mismas como entidades: con la anotación `@Entity` le indicamos a JPA que ese objeto es una entidad de la base de datos; con `@Inheritance` cómo queremos implementar la herencia en la base de dats; también se pueden declarar distintas querys con JPQL (*JP Query Language*) (Keith et al, 2018); y con las demás anotaciones sobre los atributos, añadimos características (nombre de la columna, formato de la fecha, restricción *unique, etc*), validaciones sobre los atributos (valor numérico, *string*, email, etc) y propiedades sobre las relaciones entre los distintos objetos/entidades.

Como ejemplo se muestra la clase `Empleado` muy simplificada en la figura 2.2.24.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@NamedQueries({
        @NamedQuery(name = "Empleado.listar",
                    query = "FROM Empleado"),

        @NamedQuery(name = "Empleado.buscarPorEmail",
                    query = "from Empleado e where e.email = :email")
})
public abstract class Empleado implements Serializable {

    @GeneratedValue(strategy = GenerationType.AUTO)
    @Id
    protected Long id;

    @NotBlank
    @Column(nullable = false, unique = true)
    @Email
    protected String email;

    @ManyToOne(fetch = FetchType.EAGER)
    protected Departamento departamento = new Departamento();
    .......

}
```

Figura 2.2.24. Ejemplo de la Entidad JPA Empleado

Por otro lado, para completar la configuración de JPA hace falta especificar varias de las propiedades en un fichero llamado `persistence.xml`, en el que se indican todas las entidades que se van a persistir en el sistema, así como la conexión a la base de datos, el *pool* de conexiones, etc.

Como ejemplo se muestra el fichero `persistence.xml` muy simplificado en la Figura 2.2.25.

```
<persistence ...... >
<persistence-unit name="TFG_server" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>

        <class><PAQUETES>.Entidad.Empleado</class>

        <properties>

            <!-- Base de Datos MariaDB REMOTA-->
            <property name="javax.persistence.jdbc.driver"
                      value="com.mysql.cj.jdbc.Driver"/>

            <property name="javax.persistence.jdbc.url"
                      value="jdbc:mysql://localhost/TFG_BBDD_server"/>

            <property name="javax.persistence.jdbc.user" value="..."/>
            <property name="javax.persistence.jdbc.password" value="."/>

            <!-- POOL DE CONEXIONES -->
            <property name="hibernate.c3p0.min_size" value="5"/>
            <property name="hibernate.c3p0.max_size" value="15"/>

            .......
        </properties>
    </persistence-unit>
</persistence>
```

Figura 2.2.25. Fichero de configuración de Hibernate `persistence.xml`

2.2.6 MariaDB

MariaDB es un sistema gestor de bases de datos relacionales con licencia GPL derivado de la versión 5.5 de MySQL. Reemplaza los dos motores de almacenamiento originales de MySQL, MyISAM e InnoDB, por Aria y XtraDB, que le aportan mejor rendimiento gracias a incorporar nuevas tablas para la gestión interna de los recursos. En el proyecto ha sido usada la versión del servidor MariaDB 10.1.30.

La compatibilidad con MySQL es prácticamente total, ya que se diseñó para ello, por precaución ante una posible licencia comercial de Oracle al comprar Sun Microsystem

2.2.7 Apache Tomcat

Apache Tomcat es un contenedor de servlet de código abierto desarrollado por la *Apache Software Foundation* (ASF) que permite el uso de Java servlets y páginas JSP apoyado sobre la máquina virtual de Java. En el proyecto ha sido usada la versión Apache Tomcat 8.5.23.

Tomcat Catalina proporciona la implementación de los servlets de Apache Tomcat desde la versión 4. La configuración predeterminada de Catalina se puede modificar desde los ficheros contenidos en el directorio `$CATALINA_BASE/conf`, como pueden ser los usuarios y sus permisos de acceso y configuración del servidor, opciones o valores que se aplicarán a las aplicaciones desplegadas, las rutas de carga de las clases Java, etc.

Tomcat Coyote es un componente más del ecosistema que se encarga de las conexiones soportando el protocolo HTTP 1.1 y TCP para las aplicaciones desplegadas, o para servir como servidor simple de archivos locales.

En este proyecto, Tomcat se configura para permitir el acceso usando JAAS, el control de usuarios entre aplicaciones para HTTPS, incluyendo su redirección a un puerto seguro, y se le añade un usuario más con privilegios de administración para poder realizar los despliegues desde Jenkins (explicado más adelante en el punto 2.2.3).

2.2.8 Apache Maven

Apache Maven es una herramienta de código abierto para la gestión y construcción de proyectos Java. De cara al programador toda la configuración necesaria se realiza en un fichero llamado `POM.xml` (*Project Object Model*) en el que se puede gestionar la inyección de dependencias, la compilación, documentación, herencia del proyecto, etc. En el proyecto se ha usado la versión Apache Maven 3.5.2.

Todo ello complementado con una suite de comando muy extensa con la que poder gestionar completamente el ciclo de vida del proyecto, que consta de ocho fases (cada fase incluye lo anteriores):

1. `Clean`: Limpia la carpeta target de las ejecuciones anteriores.
2. `Validate`: Valida el código del proyecto.
3. `Compile`: Compila el código fuente del proyecto.
4. `Test`: Ejecuta los test configurados.
5. `Package`: Genera el artefacto del proyecto.
6. `Verify`: Verifica el artefacto creado.
7. `Install`: Instala el artefacto en el repositorio local.
8. `Deploy`: Sube el artefacto a un repositorio Maven remoto.

Ambas aplicaciones de este proyecto han sido creadas con la estructura de directorios específica para un proyecto Maven básico. Tal y como especifica la Figura 2.2.26.

- En la raíz del proyecto se localiza el fichero `pom.xml` con toda la configuración.
- El directorio `target` contiene los resultados de compilaciones anteriores.
- Del directorio `src` cuelgan test donde se alojan todos los test que se ejecutarán y `main` con el resto de ficheros:
    - Un directorio `main/java/META-INF` en el que está el fichero de configuración de Hibernate.
    - Y en `main/java/com.rodrigo.TFG_server` está contenido todo el código Java de la aplicación.
    - En `main/resources` el fichero de configuración de SLF4J que hereda el de Log4j.
    - En `main/webapp` los ficheros de `web.xml` y `cxf-servlet.xml` para configurar la aplicación web.

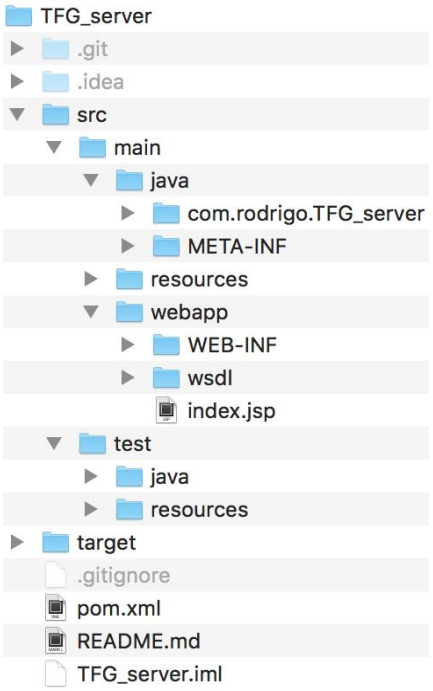

Figura 2.2.26. Jerarquía de directorios Maven

2.3 Tecnologías de prueba y de control de versiones

2.3.1 JUnit

JUnit es un marco destinado a la realización de pruebas unitarias de aplicaciones Java. Permite ejecutar las clases Java del proyecto en un entorno virtualizado y controlado para poder evaluar si los distintos métodos de dicha clase realizan las funciones esperadas. En el proyecto se ha usado la versión JUnit 5.0.2.

Su funcionamiento es muy sencillo: se crean una clase test para cada clase que se desea probar del proyecto y después se crean métodos test que crearán distintos entornos para probar los métodos de la clase objetivo de distintas formas mediante el uso de aserciones binarias de valor esperado y valor recibido. De ser correctas, JUnit nos retorna un resultado satisfactorio, y de no serlo, nos indica que valor resultó erróneo, y cuál se esperaba recibir.

JUnit 5 utiliza varias funciones para hacer las pruebas:

- `fail()`: permite que falle cuando hay éxito.
- `assertTrue()`: comprueba que una condición booleana es cierta.
- `assertFalse()`: comprueba que una condición booleana es falsa.
- `assertEquals()`: comprueba que dos valores coinciden.
- `assertNull()`: comprueba que un objeto es null.
- `assertNotNull()`: comprueba que un objeto no es null.
- `assertSame()`: comprueba si dos variables se refieren al mismo objeto.
- `assertNotSame()`: comprueba si dos variables no se refieren al mismo objeto.

2.3.2 GitHub

GitHub es una plataforma online para alojar proyectos software utilizando el sistema distribuido de control de versiones Git.

Para este proyecto se han destinado dos repositorios en la cuenta personal del alumno, uno para cada aplicación. La para gestión del software se ha optado por tener una rama *Master* y otra *Development*. En la primera se encuentra el código listo para ejecutar y desplegar que ha pasado todas las pruebas. En la segunda se aloja el código que está en proceso de desarrollo y que por tanto aun no pasa todos los test realizados.

Para la gestión del repositorio se ha usado el programa SourceTree que facilita enormemente las tareas de administración al proveer de una interfaz gráfica muy intuitiva.

A lo largo del desarrollo del proyecto se ha mantenido la filosofía de la Integración Continua de código, es decir, en cada "jornada" de desarrollo se realizan las subidas al repositorio de los cambios modificados de forma que, de haber algún error, test fallido, o modificación necesaria es mucho más sencillo modificar dicho commit.

2.3.3 Jenkins

Jenkins es un servidor de código abierto para la automatización de tareas y *pipelines* de proyectos software que proporciona una solución de Integración Continua (CI) y entrega continua. Soporta multitud de complementos instalables a modo de *plugins*, como son uso: complementario con CVSs como Git, notificaciones de resultados por *email*, integración con Maven, ejecuciones de trabajos con *pipelines*, etc. En el proyecto ha usado la versión del servidor Jenkins 2.132.

En este proyecto, Jenkins ha sido configurado para simular un entorno real de desarrollo de código, ya sea de un equipo pequeño o de un proyecto muy grande, que se guía por la filosofía de integración y entrega continua, es decir: verificación de código, gestión y ejecución de pruebas, mantenimiento, compilación y despliegue en Apache Tomcat de forma automática y sin la necesidad de intervenir en el proceso, salvo en la subida de código nuevo desarrollado a GitHub.

Para este proyecto se han creado dos *jobs* de proyecto Maven, uno para cada aplicación, que realizan el mismo procedimiento.

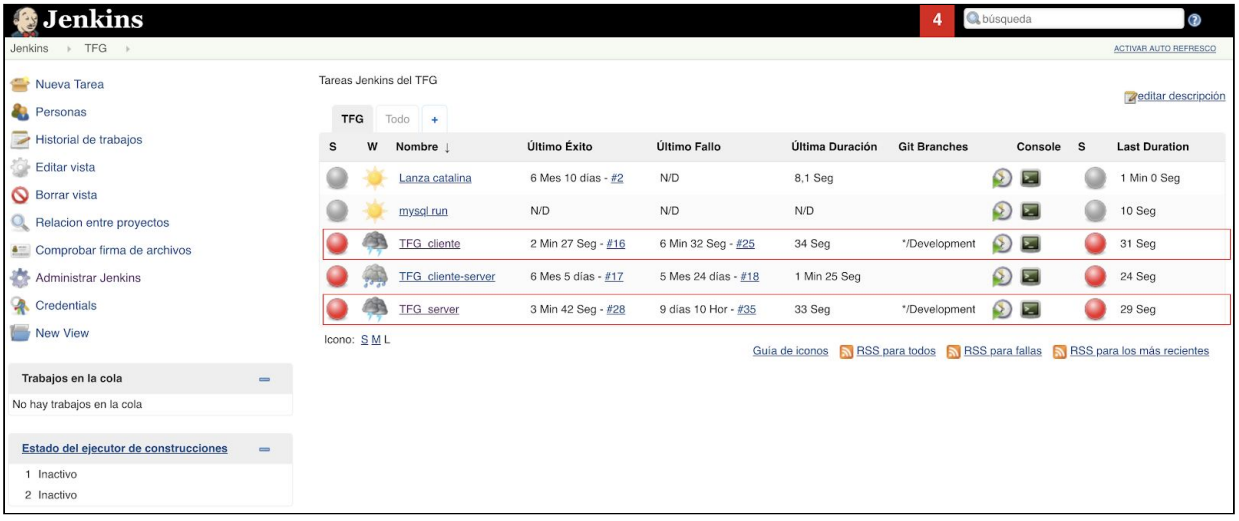

Figura 2.3.1. Panel principal de Jenkins

La configuración de las tareas tiene tres fases: procedimientos para la preparación del entorno, ejecución de los comandos especificados y acciones posteriores al *build* y pruebas.

- Preparación del entorno:
  - *GitHub*: Para empezar, se configura el acceso al repositorio de Github, usando la URL y las credenciales de acceso (si no se han configurado antes se agregan nuevas, de forma que Jenkins se encarga de guardarlas de forma segura) y se selecciona la rama de la que cogerá el código, en este caso la rama *Development*, como se puede ver en

la figura 2.3.2. Jenkins realiza un *pull* completo pisando el código anterior de la rama en un directorio interno al servidor para la realización de la tarea.

- *Trigger de ejecución*: se configura cada cuanto tiempo se quiere consultar nuevos cambios en el repositorio usando expresiones CRON (Tansley, 1999). De haber un nuevo cambio descarga el código.

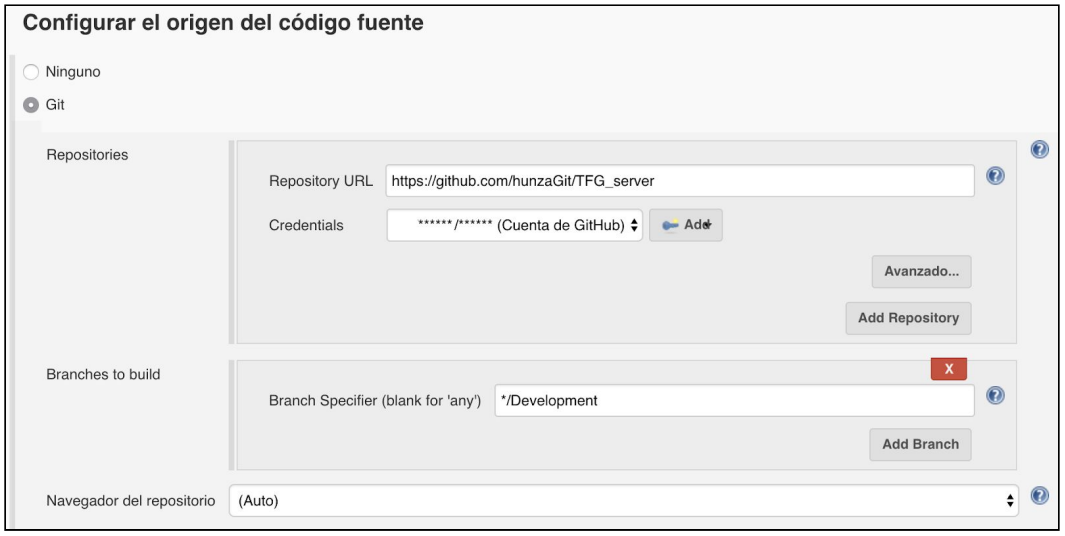

Figura 2.3.2. Configuración del job para coger el código de GitHub.

- Ejecución de comandos:
  - *Ejecuciones Maven*: Jenkins permite ejecutar tareas Maven de nivel superior (las fases vistas anteriormente) sobre el proyecto. En este caso, usando la versión Maven 3.5.2, se configura para que ejecute primero un `clean` limpiando los posibles restos de ejecuciones anteriores, posteriormente se un `package`. Esto creará un artefacto del código, lo que incluye, entre otros, la ejecución de todos los test (esto tiene especial importancia que veremos en los pasos siguientes). Mostrado en el Figura 2.3.3.

- Acciones posteriores:
  - Llegados a este punto puede haberse dado dos casos: que todas las ejecuciones hayan sido satisfactorias (compilación, test pasados correctamente, etc.), o que alguno de los pasos haya fallado, ya sea por código erróneo o por ejemplo unos test que han retornado errores.

Figura 2.3.3. Ejecuciones Maven del job en Jenkins

Si la ejecución hasta ahora ha sido satisfactoria se realizan dos pasos más:

- *Git Publisher*: Si todo ha ido correctamente significa que el código probado es estable, por lo que está listo para subirse a otra rama del repositorio. En este caso a la rama *master* (lo correcto sería a una dedicada a preproducción o calidad, pero debido a ser un proyecto unipersonal se optó por no hacerlo de esta forma). Tiene varias opciones como un merge automático, forzar el *push*, añadir *Tags* o notas.

- *Deploy WAR en Apache Tomcat*: tras el paso anterior y habiendo asegurado un nuevo *commit* en la rama de código estable, se procede a desplegar la aplicación en Apache Tomcat. Para ello se elige el fichero WAR resultante de la ejecución Maven, que por la configuración del `pom.xml`, es `target/NOMBRE_PROYECTO.war`, se selecciona el *context path* bajo el que se publicará en el servidor y añadiendo un contenedor (Apache Tomcat 8.5 en este caso) se ingresan las credenciales creadas para Jenkins en (en `$CATALINA_BASE/conf/tomcat-users.xml`) y la URL bajo la que está operando Tomcat. Tras esto queda disponible la aplicación web en la URL destinada por Tomcat.

Si alguna de las ejecuciones anteriores, por la razón que sea, falla se ha configurado como extra la posibilidad de que se envíen *emails*:

- Notificación por email: En este caso hace falta configurar dos cosas previas, configurar el servidor de correo saliente, en este caso el SMTP de gmail para el correo institucional de la UCM y de ser más miembros en el desarrollo, crear cuentas de usuario de Jenkins y enlazarlas a la cuenta de GitHub que realizó el *commit* que rompió el proyecto. Para este proyecto solo se ha indicado un único destinatario.

Tras todo esto se tiene un entorno completo de Integración Continua que permite el desarrollo y entrega de *software* de forma ágil y cómoda, y unido a subidas de código pequeñas y periódicas, se puede crear un código mantenible que reduce enormemente la deuda técnica acumulada.

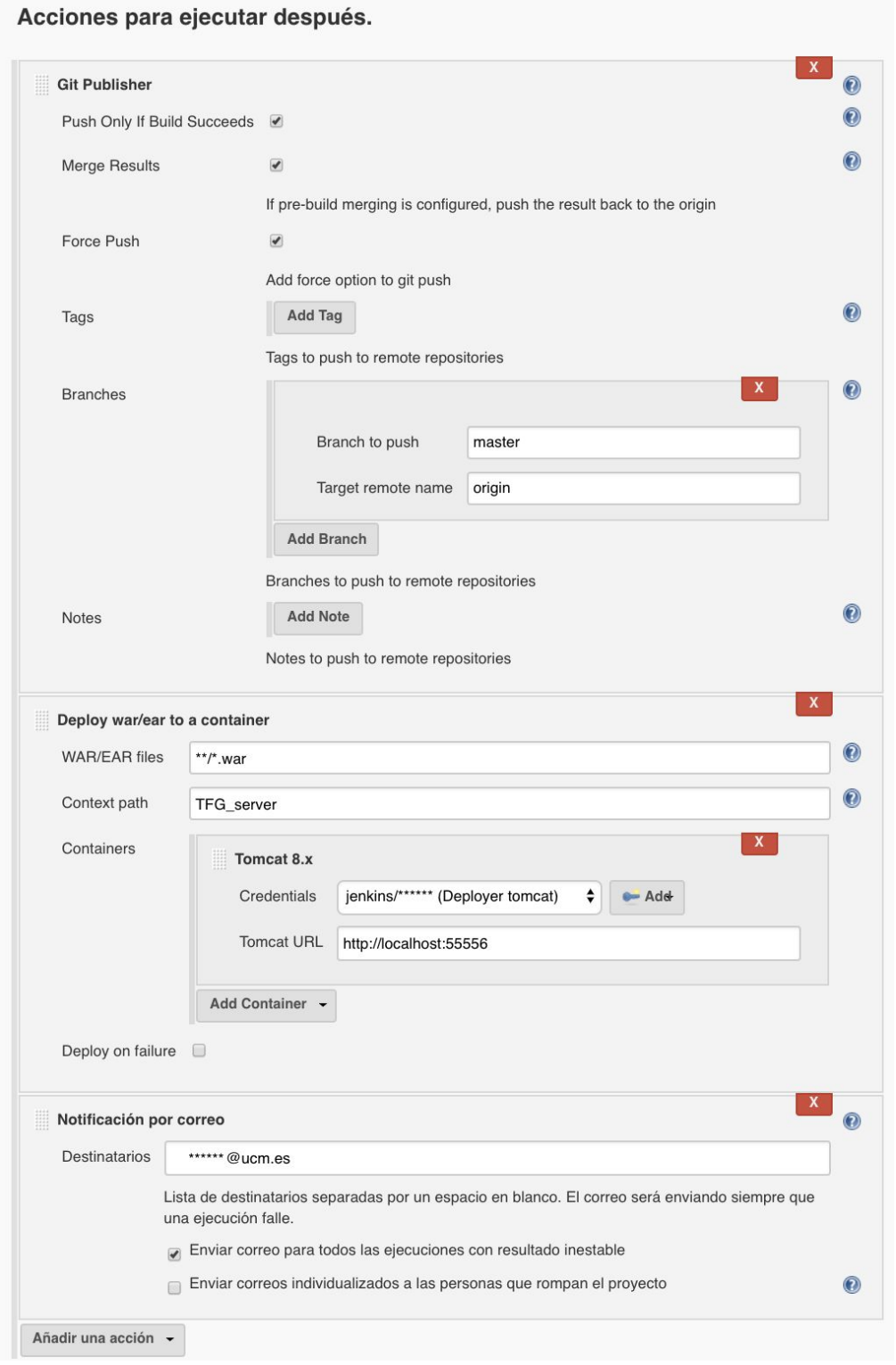

Figura 2.3.4. Configuración de Git Publisher, Deploy en Tomcat y notificación de email.

Capítulo 3. Requisitos de la aplicación

La finalidad de este TFG no es crear una aplicación grande con muchos casos de uso, sino combinar muchas tecnologías actuales en un ecosistema estable, funcional y escalable con una arquitectura basada en patrones multicapa. Además, este TFG está desarrollado por un único alumno. Por tanto, el dominio de la aplicación se ha mantenido muy manejable, solo con tres entidades con las que trabajar, incluyendo relaciones 1 a N y M a N y especialización. El dominio se centra en la gestión de departamentos, empleados y proyectos en una organización.

3.1 Modelo del dominio

La Figura 5 ilustra el modelo del dominio de la aplicación con un diagrama de clases UML.

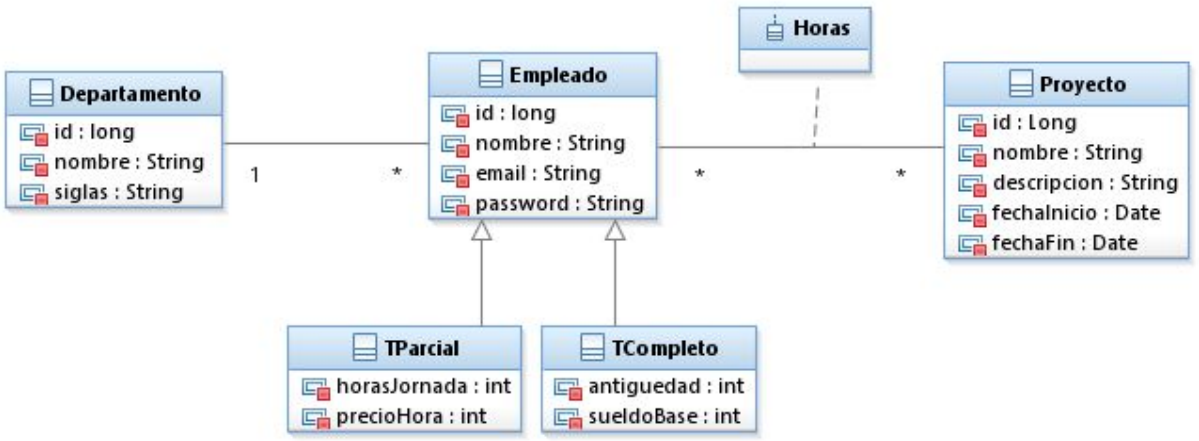

Figura 3.1.1. *Modelo del Dominio*

A continuación, se describen las funciones asociadas a cada entidad:

- Gestión de departamentos. Gestión CRUD (*Create, Read, Update, Delete*) de departamentos, y cálculo de la nómina de los departamentos como suma de las nóminas de sus empleados.
- Gestión de empleados. Gestión CRUD (*Create, Read, Update, Delete*) de empleados.
- Gestión de proyectos. Gestión CRUD (*Create, Read, Update, Delete*) de proyectos. También se encarga de la gestión de asignaciones de empleados a proyectos.

3.2 Requisitos funcionales

En este proyecto concurren una serie de factores atípicos sobre los requisitos: son muy sencillos, la persona responsable de especificarlos es la misma que va a hacer el diseño, el equipo de desarrollo está formado por una persona, y el dominio es familiar al responsable del proyecto. Por tanto, se ha procedido a dar una descripción muy simple de los mismos, asimilable a los requisitos de usuario.

Se supone que todas las entidades son gestionadas por un identificador único (ID) que es asignado automáticamente por la aplicación cuando son creadas.

3.2.1 Departamento

- **Crear**: registrar un departamento en el sistema. No puede existir dos departamentos con las mismas siglas.
- **Mostrar por ID**: Muestra la información completa de un departamento buscando por ID.
- **Mostrar por Siglas**: Muestra la información completa de un departamento buscando por siglas
- **Actualizar**: Actualiza la información de un departamento en el sistema.
- **Eliminar**: Elimina un departamento si no tiene empleados asignados.
- **Listar**: Lista la información básica de cada departamento.
- **Calcular Nómina**: Calcula la nómina en función de los empleados asignados (incluido en los casos de uso de mostrar departamento).

3.2.2 Empleado

- **Crear**: registra un empleado de tipo “tiempo completo” o “tiempo parcial” en el sistema. No pueden existir dos empleados con el mismo email.
- **Mostrar por ID**: Muestra la información completa de un empleado buscando por ID.
- **Mostrar por email**: Muestra la información completa de un empleado buscando por email.
- **Actualizar**: Actualiza la información de un empleado en el sistema.
- **Eliminar:** Elimina un empleado del sistema, eliminando sus asignaciones a proyectos.
- **Listar**: Lista la información básica de cada empleado.

3.2.3 Proyecto

- **Crear**: Registra un proyecto en el sistema. No pueden existir dos proyectos con el mismo nombre.
- **Mostrar por ID**: Muestra la información completa de un proyecto buscando por ID.
- **Mostrar por nombre**: Muestra la información completa de un proyecto buscando por nombre.
- **Actualizar**: Actualiza la información de un proyecto en el sistema.

- **Eliminar**: Elimina un proyecto del sistema, eliminando sus asignaciones.
- **Listar**: Lista la información básica de cada proyecto.
- **Asignar empleado a proyecto**: Crear una relación entre un empleado y un proyecto, guardando el número de horas asignadas.

3.3 Requisitos no funcionales

3.3.1 Seguridad

La seguridad del proyecto consta de dos partes, una referida al *front-end* y otra referida al acceso a los servicios web del *back-end*.

- *Control de acceso front-end:* Para controlar el acceso a los recursos del sistema se requiere realizar un login previo con el email y contraseña del usuario de la aplicación cliente. Este control se delegará en JAAS. Para poder acceder a los recursos además del login es necesario poseer el rol adecuado a las funciones que se quieren realizar:
    - El rol *USUARIO* tiene permiso de acceso a los recursos bajo la URI "`/views/Modulo_Usuario/*`" que contiene las funciones de muestra de datos.
    - El rol *ADMIN* tiene acceso a los recursos bajo la URI "`/views/Modulo_Administrador/*`" que contiene todas las funcionalidades implementadas.
    - El rol *SUPERUSER* tiene acceso a ambos módulos.

- *Control de acceso a servicios web:* todos los servicios web, tanto los expuesto mediante JAX-WS como con JAX-RS, están bajo el protocolo HTTPS y accesibles bajo usuario y contraseña. Para ello es necesario:
    - Crear un *keystore* con un certificado (autofirmado en este caso) y se ha añadirlo a la configuración de Tomcat.
    - Crear una base de datos con usuario y contraseña para que Tomcat pueda verificar los accesos mediante un nuevo *Realm* en el `server.xml`.
    - Indicar en el fichero `web.xml` de la aplicación servidor bajo qué URL se quiere restringir el acceso por HTTPS.
    - Añadir en el cliente las líneas de código para el envío del usuario y contraseña.

3.3.2 Arquitectura, Tecnologías y Aplicaciones

Es un requisito utilizar una arquitectura multicapa, tal y como se ha descrito en la sección 2.1. Por lo mencionado en la introducción, también consta como requisito el uso distintas tecnologías y aplicaciones:

- Capa de presentación implementada con JSF utilizando la librería Mojarra. Las restricciones de seguridad para el acceso de usuario están gestionadas con JAAS, que limita el acceso a los usuarios (explicado en el apartado 3.3.1).
- Los servicios web se expondrán con JAX-WS para los módulos Empleado y Proyecto, mediante el protocolo SOAP y JAX-RS para exponer el módulo Departamento usando REST (con datos intercambiados en formato XML). En ambos casos se usa la librería Apache CXF.
- La capa de integración (acceso a recursos) será implementada con JPA, usando la librería Hibernate.
- Se usará como SGBDR MariaDB.
- Las pruebas se llevarán a cabo con JUnit.
- La gestión de librerías se llevará a cabo con Apache Maven.
- El sistema control de versiones usado será GitHub. El código del proyecto está disponible en:
    - Servidor: https://github.com/hunzaGit/TFG_server
    - Cliente: https://github.com/hunzaGit/TFG_cliente
- La Integración Continua se gestionará con Jenkins.

También es requisito, y de gran importancia, que se gestionen las transacciones en los accesos a los datos, así como su uso concurrente, evitando los problemas asociados a éste. En cuanto al número de usuarios simultáneos, esta es una cuestión de despliegue de la que se prescinde en este trabajo.

Capítulo 4. Capa de presentación y negocio de la aplicación cliente

4.1 Interfaz gráfica de usuario

Se ha intentado diseñar una capa de presentación sencilla y funcional. Así, la aplicación web cliente consta de cuatro páginas diseñadas desde cero: la de inicio, la de *login*, la página de contenido que pueden ver los usuarios no administradores, y la página de contenidos que pueden ver los usuarios administradores. En todas las páginas se ha seguido el patrón de diseño dado por Bootstrap 3.3.7 (Twitter, 2011). Bootstrap es un *framework* de desarrollo web liberado por Twitter que tiene como objetivo facilitar el diseño de interfaces *responsive*.

La página de inicio es muy sencilla ya que sólo sirve para dar acceso al resto de páginas. Consta del título del trabajo, dos enlaces para el acceso a las funciones de usuario y de administración, y un último enlace con el que se puede cerrar la sesión activa. La Figura 4.1.1 la describe.

Figura 4.1.1. *Página de inicio*

Tal y como describe la Figura 4.1.2, la página de *login* consta únicamente de un pequeño formulario en el que introducir el email y la contraseña del usuario, el botón de acceso y un acceso a la página de inicio. Se muestra un pequeño mensaje de error en caso de que el falle la validación.

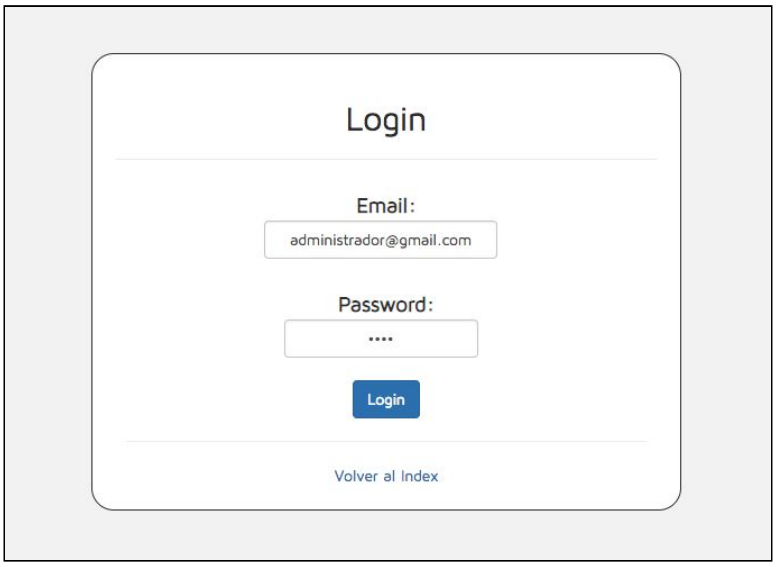

Figura 4.1.2. Página de login

Una vez registrados, se accede a la página del usuario correspondiente (administrador o no). Ambas páginas están diseñadas igual, usando el mismo patrón de diseño y colores. Las páginas están diseñadas con plantillas JSF siguiendo el estilo SPA (*Single Page Application*) (Scott, 2015) en el que toda la funcionalidad está en la misma página, sin tener que estar navegando de una a otra página de forma que al usuario le resulta una navegación más sencilla. La Figura 4.1.3 muestra la página para administradores.

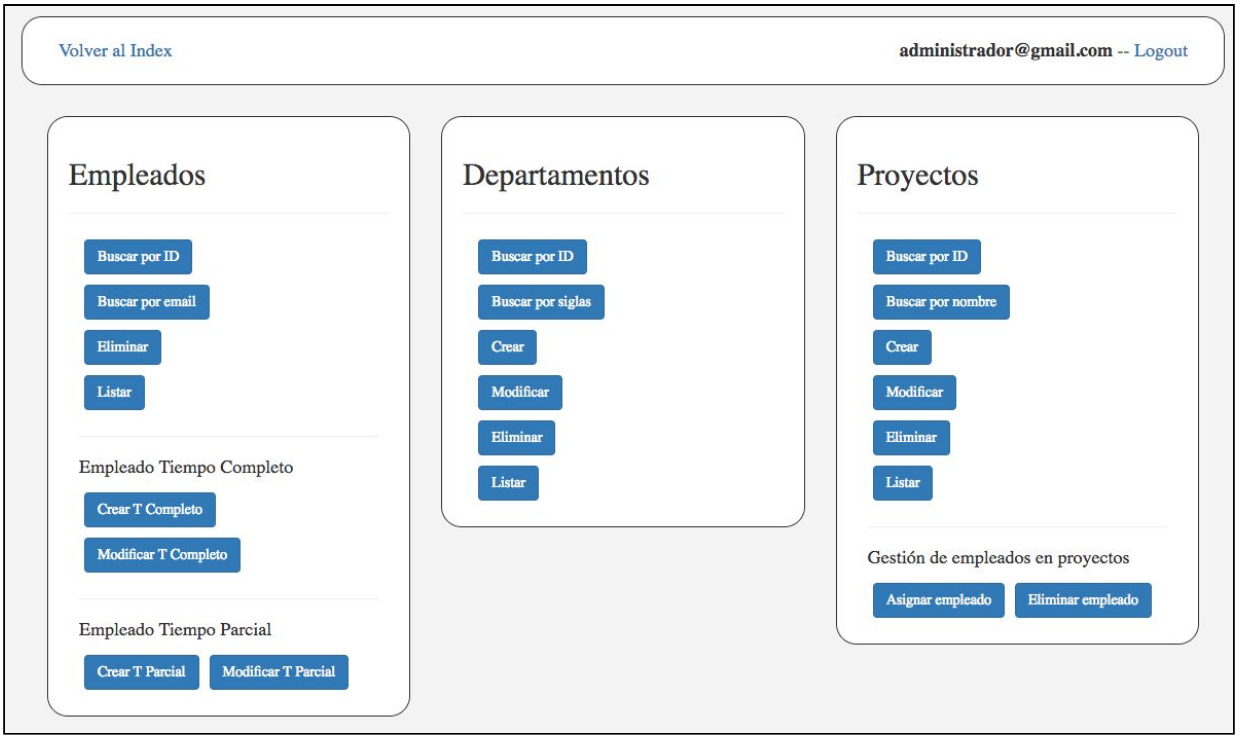

Figura 4.1.3. Página del administrador

A cada una de las funciones disponibles en el sistema se accede seleccionando los botones, encuadrados dentro de cada módulo. Al hacerlo aparece debajo el formulario correspondiente a la acción que se ha elegido, y tras rellenarlo se despliega otro cuadro con toda la información solicitada. La Figura 4.1.4 muestra el ejemplo de buscar un departamento por siglas y la Figura 4.1.5 muestra el ejemplo de asignar un empleado a un proyecto y unas horas a dicha relación.

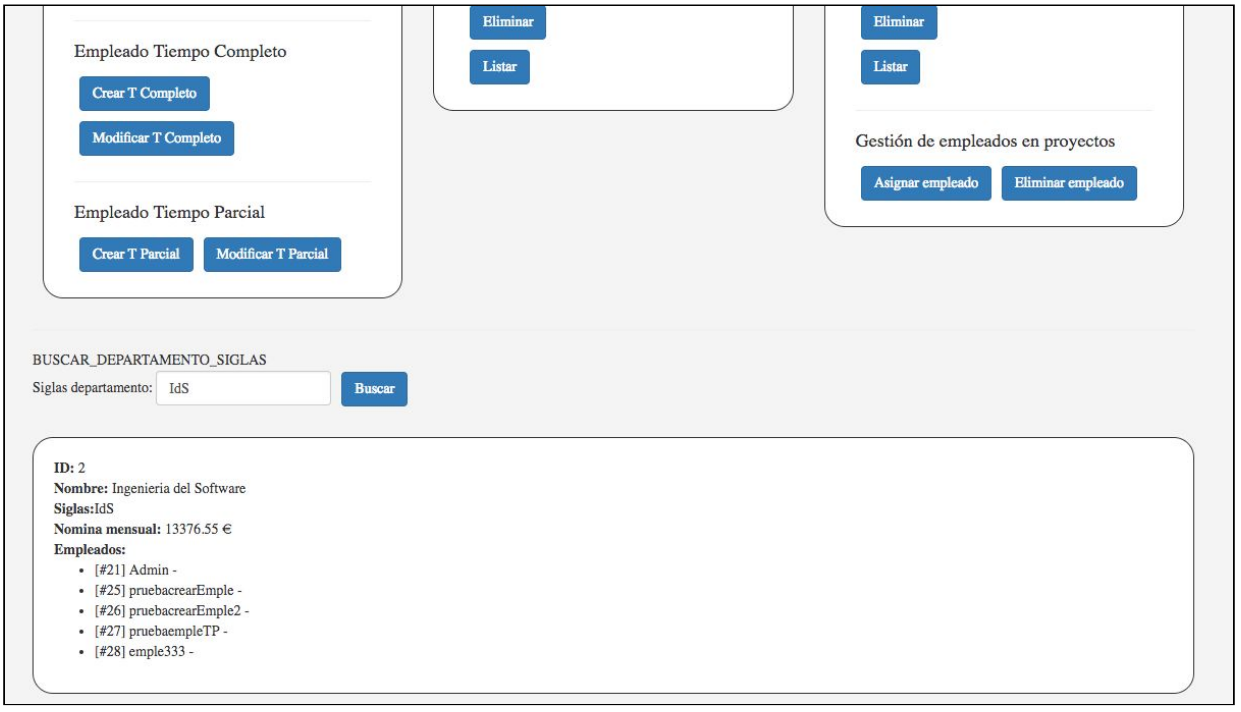

Figura 4.1.4. Página del administrador - buscar Departamento por Siglas.

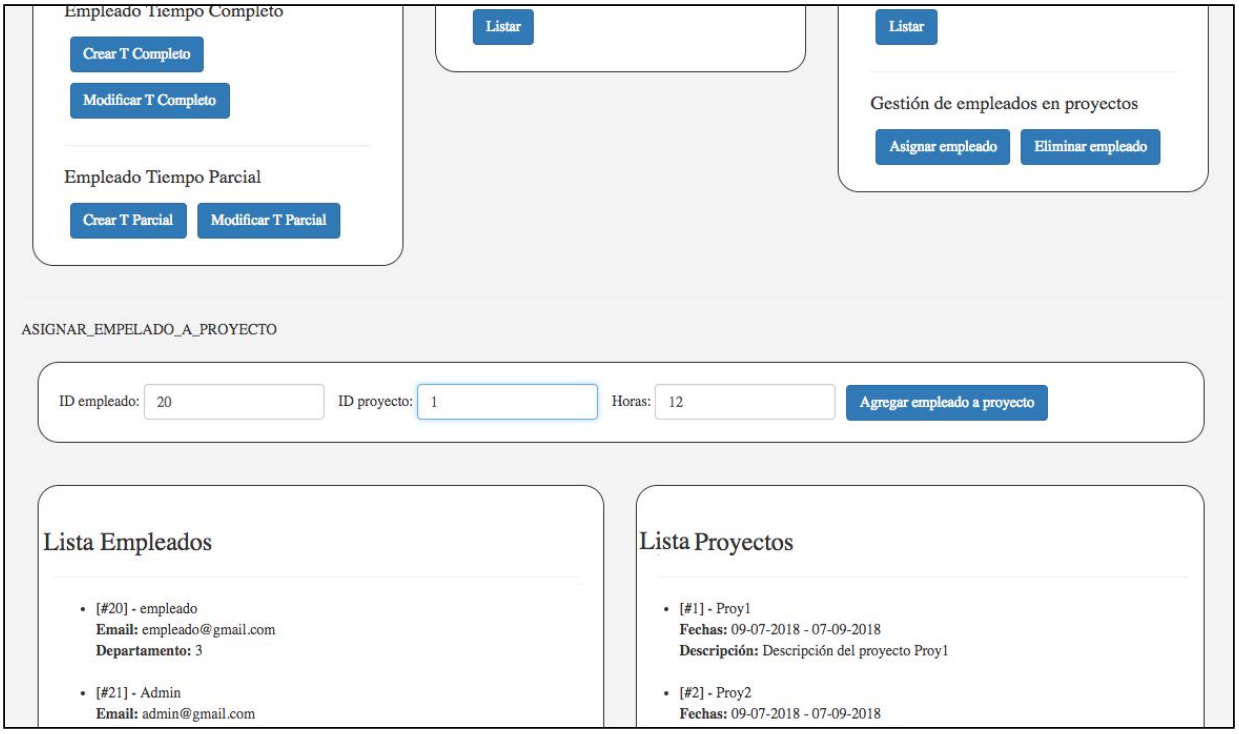

Figura 4.1.5. Página del administrador - Asignar empleado a proyecto

Las interfaces al estar diseñadas con Bootstrap son *responsive* por lo que la versión móvil es adaptativa, en la que los distintos marcos que ocupan la interfaz se reordenan automáticamente según se reduce o amplía el tamaño de pantalla.

Como se puede ver en las figuras 4.1.6 y 4.1.7, al reducir el tamaño de pantalla, los marcos de izquierda a derecha se reordenan para quedar de arriba a abajo.

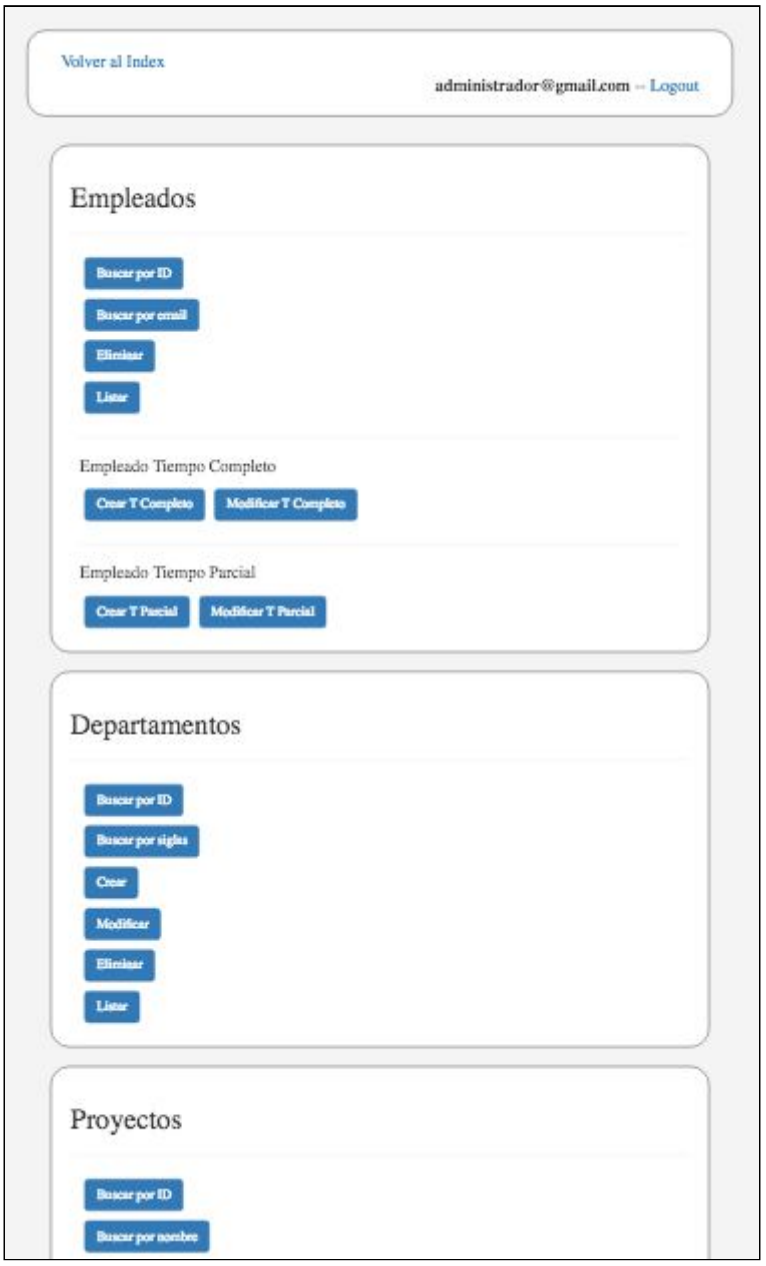

Figura 4.1.6. Página del administrador versión móvil

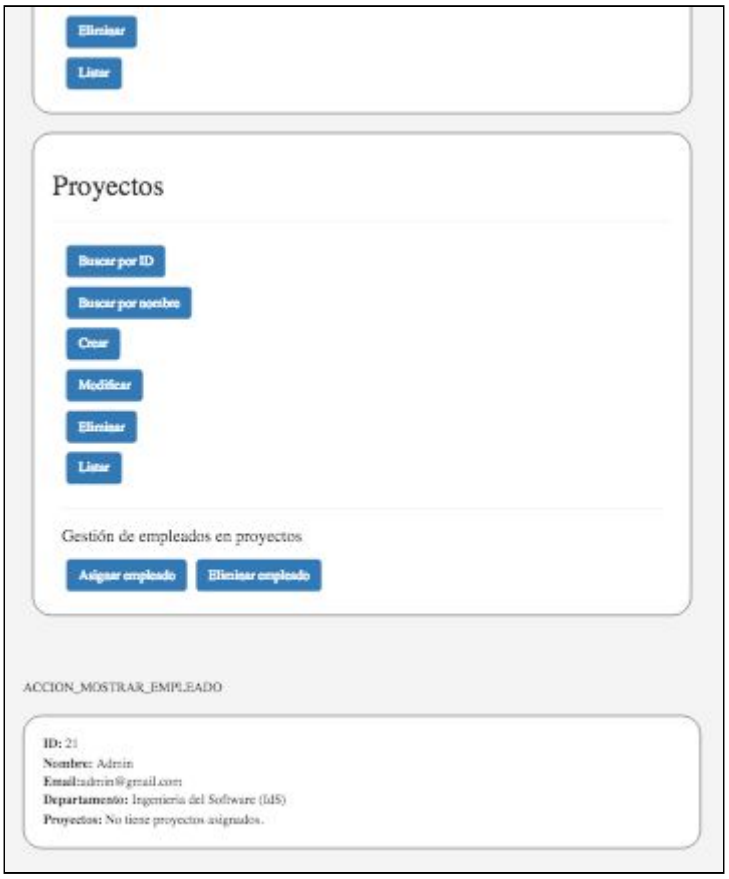

Figura 4.1.7. Página del administrador versión móvil - buscar Departamento por Siglas.

4.2 Principales patrones utilizados e implementación con tecnologías de desarrollo en la capa de presentación de la aplicación cliente

Como hemos visto la interfaz es muy sencilla. Cada uno de los formularios XHTML que hay que rellenar para las distintas acciones está conectado, gracia a JSF, con clases Java, *managed beans* (el correspondiente a cada módulo). Cada uno de los campos del formulario está representado mediante los atributos del *managed bean*.

JSF está basado en el patrón modelo-vista-controlador, donde el *controlador de aplicación* (Alur et al,. 2003) utiliza comandos que incluyen los datos de las vistas como atributos suyos: los *managed beans*.

Una vez que la petición es enviada la aplicación la procesa (explicado más abajo). Cuando ha terminado, JSF genera la página HTML con los nuevos datos solicitados o algún mensaje de error o aviso en caso de la acción no haya terminado satisfactoriamente.

Las figuras 4.2.1-4.2.3 muestran diagramas de clase UML (Miles, 2006) que explican la estructura de los *managed bean* de los tres módulos desarrollados.

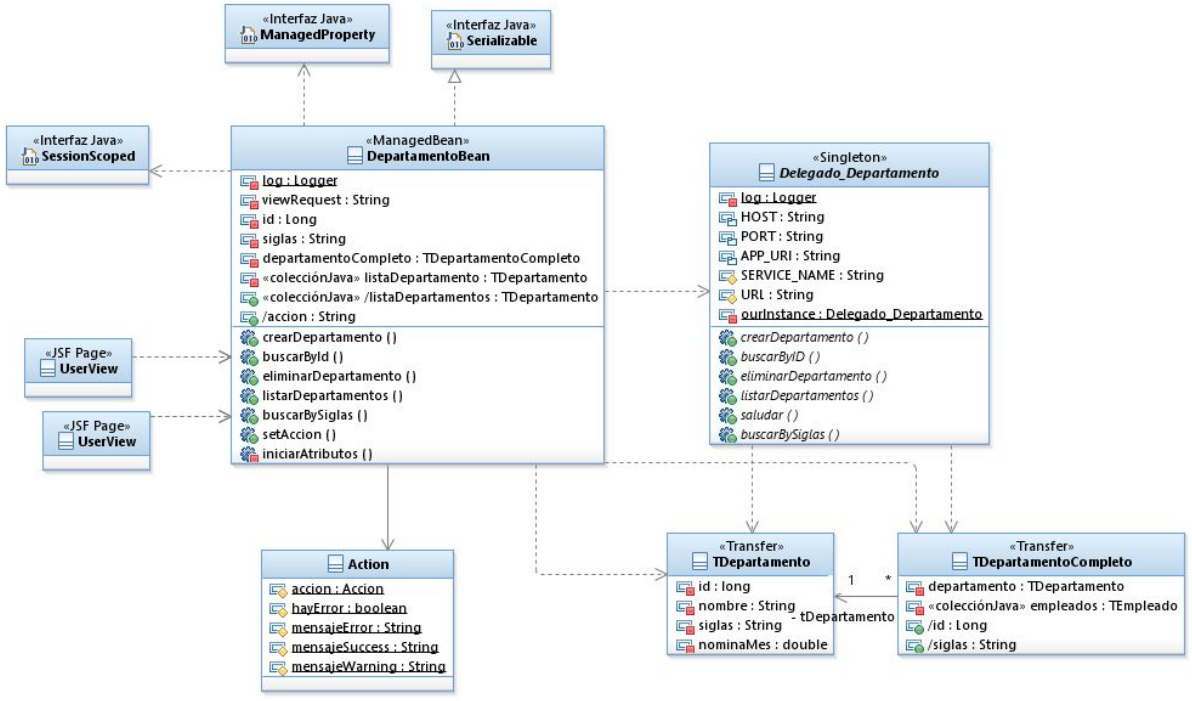

Figura 4.2.1. Diagrama de clase de managed bean de Departamento.

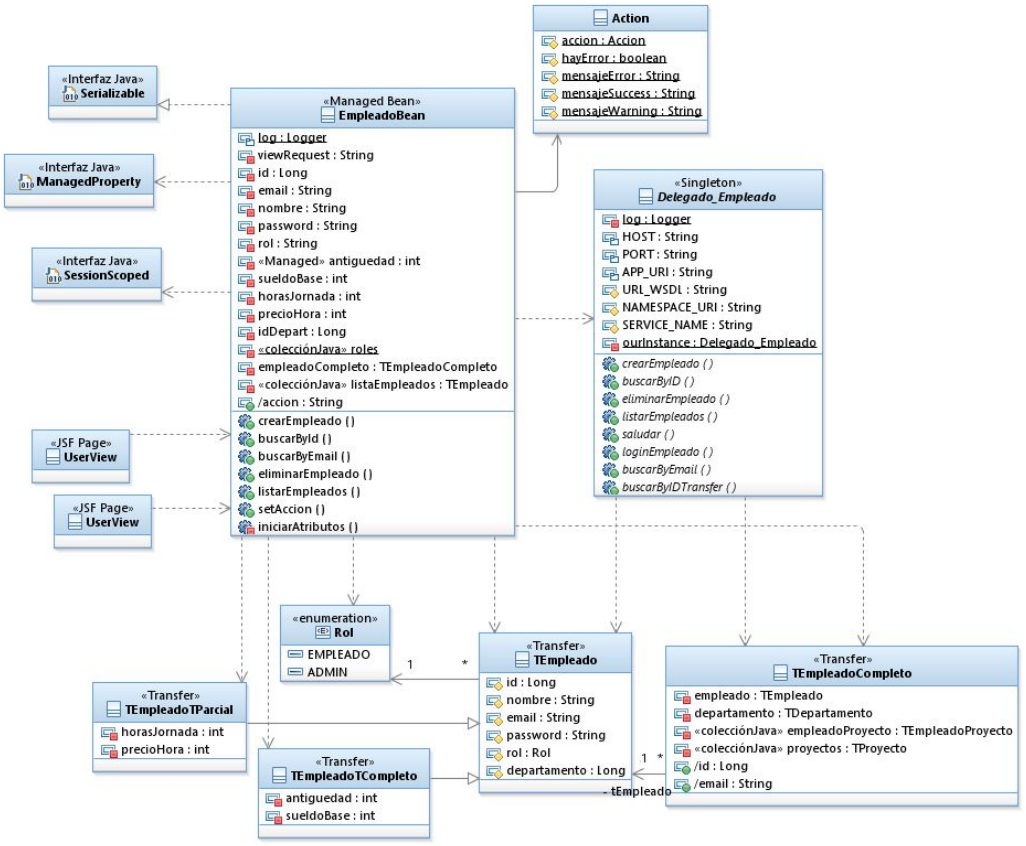

Figura 4.2.2. Diagrama de clase de managed bean de Empleado

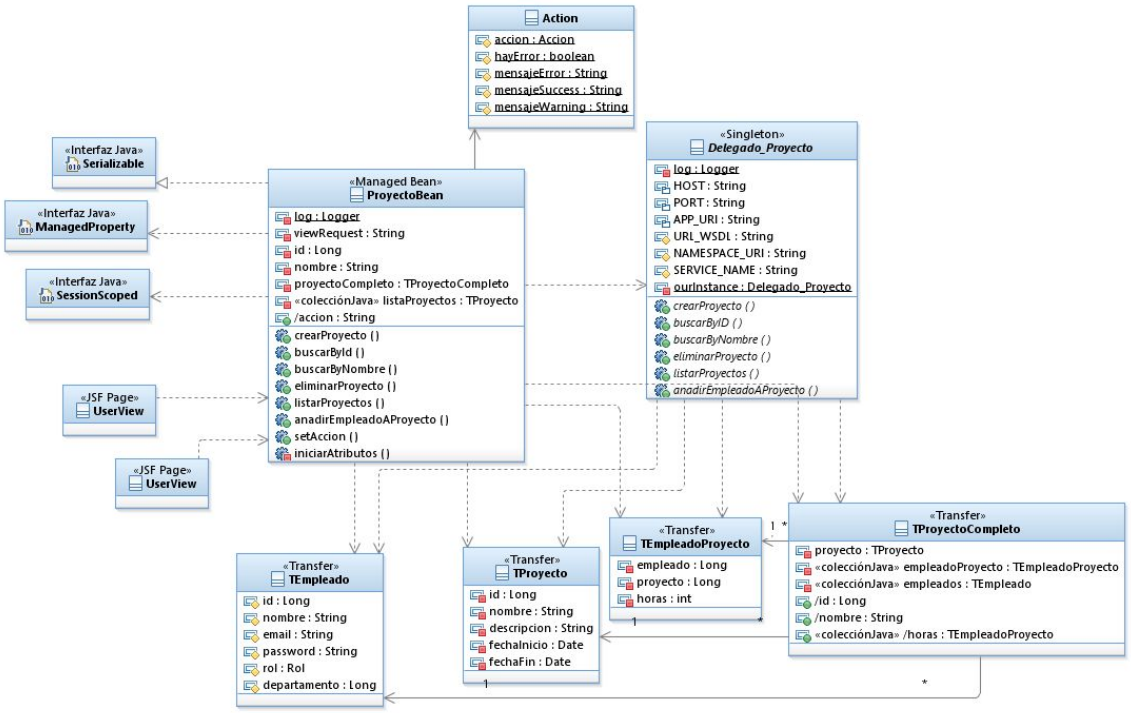

Figura 4.2.3. Diagrama de clase de managed bean de Proyecto

Como se puede observar, los managed bean se conectan con la capa de negocio a través de un *delegado del negocio* (Alur et al., 2003), específico para cada módulo, e implementado como singleton (Gamma et al., 1994).

No se proporciona una descripción de la estructura navegacional de la aplicación porque conllevaría usar extensiones a UML como (Conallen, 2002) o (Humberto & Navarro, 2017), lo cual complicaría aún más la realización de este trabajo.

Una vez definida la estructura de clases de la capa de presentación, se puede pasar a los diagramas de secuencia, con los que se puede ver de forma detallada el comportamiento de los métodos implementados en el proyecto. Se han elegido dos requisitos, *Buscar Departamento por ID* y *Crear un Empleado* para mostrar cómo funciona la capa de presentación. Este comportamiento se puede extrapolar al resto de casos de uso del sistema. La Figura 4.2.4 muestra un diagrama de secuencia del comportamiento del *managed bean* de departamento, y la Figura 4.2.5 muestra un diagrama de secuencia del comportamiento del *managed bean* de empleado. A pesar de que la implementación del primer caso de uso recae en un servicio web REST, y la segunda en un servicio web SOAP, como se puede ver en ambos diagramas, la presencia de delegados del negocio oculta este hecho a la capa de presentación.

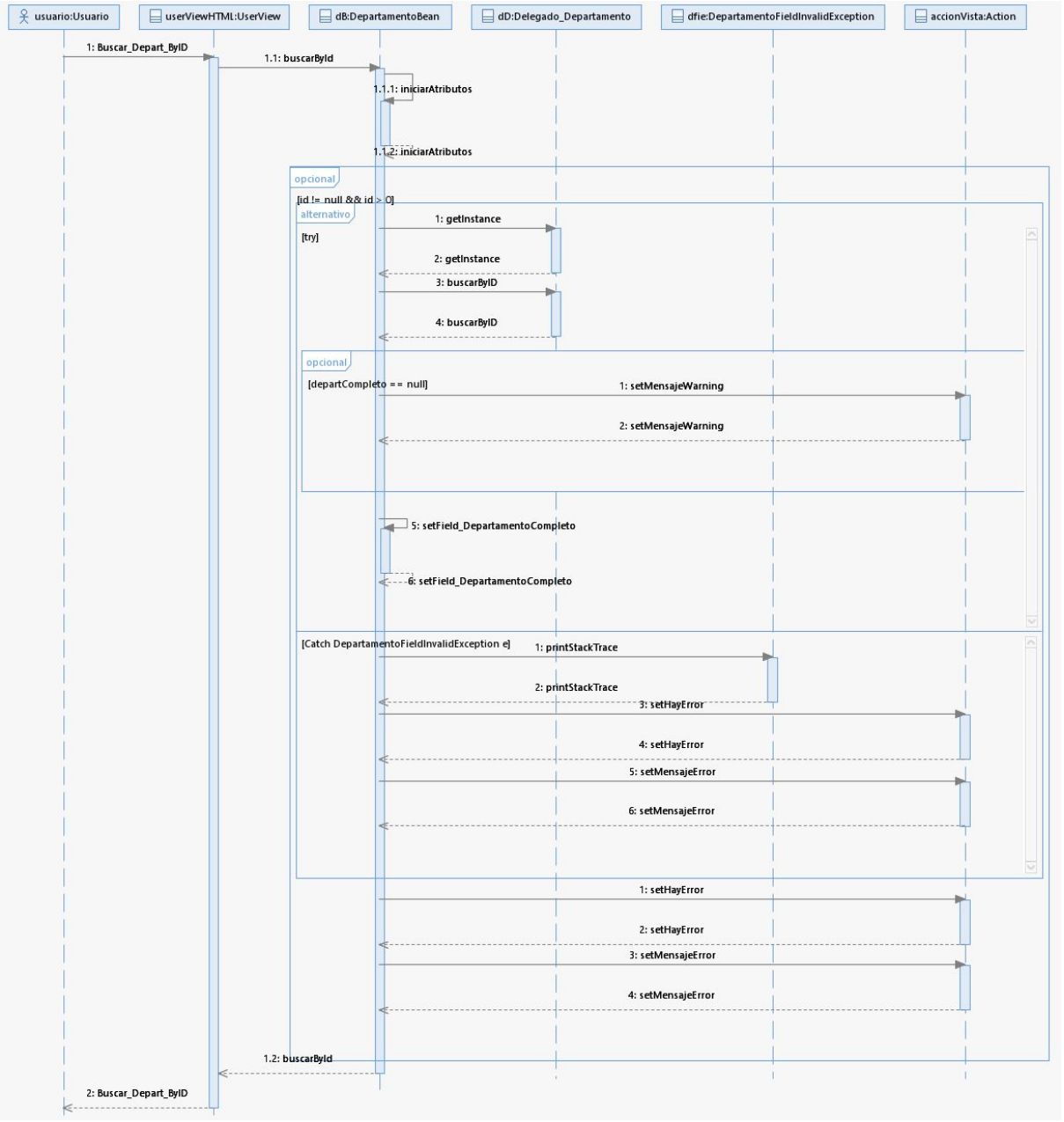

Figura 4.2.4. Diagrama de secuencia de buscar Departamento por ID en `DepartamentoBean`

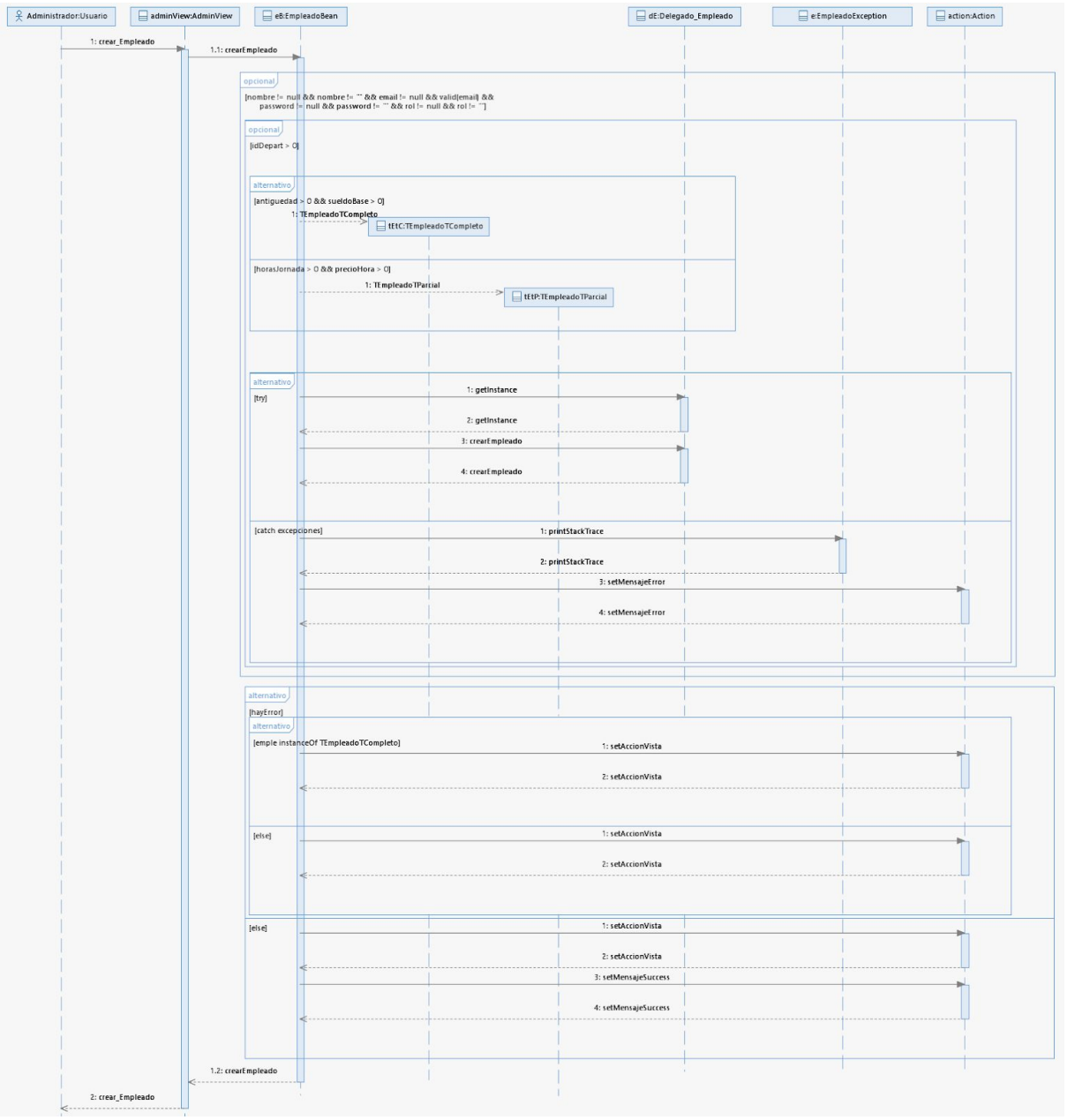

Figura 4.2.5. Diagrama de secuencia de Crear Empleado en `EmpleadoBean`

4.3 Principales patrones utilizados e implementación con tecnologías de desarrollo en la capa de negocio de la aplicación cliente

La capa de negocio de la aplicación cliente está basada en tres delegados del negocio que ocultan los *proxies* (Alur et al., 2003) de acceso a los servicios web.

En el caso del delegado de Departamento implementa las conexiones con los servicios REST usando JAX-RS (Figura 4.3.1) y los delegados de Empleado y Proyecto usando JAX-WS para los servicios SOAP (Figuras 4.3.2 y 4.3.3).

En el caso de los servicios web REST, la invocación de los servicios recae en las clases JAX-RS `Client`, `WebTarget` y en el interfaz `Invocation.Builder`, cuya implementación es la responsable de la invocación REST mediante protocolo GET, POST, PUT o DELETE. Los datos se serializan en un formato concreto (por ejemplo, XML, JSON, etc.) a través de la clase JAX-RS `Entity`.

En el caso de los servicios web SOAP, la invocación de los servicios recae en la clase Java `Service`, que genera el proxy de acceso (Alur et al., 2003) responsable de la invocación mediante mensajes SOAP.

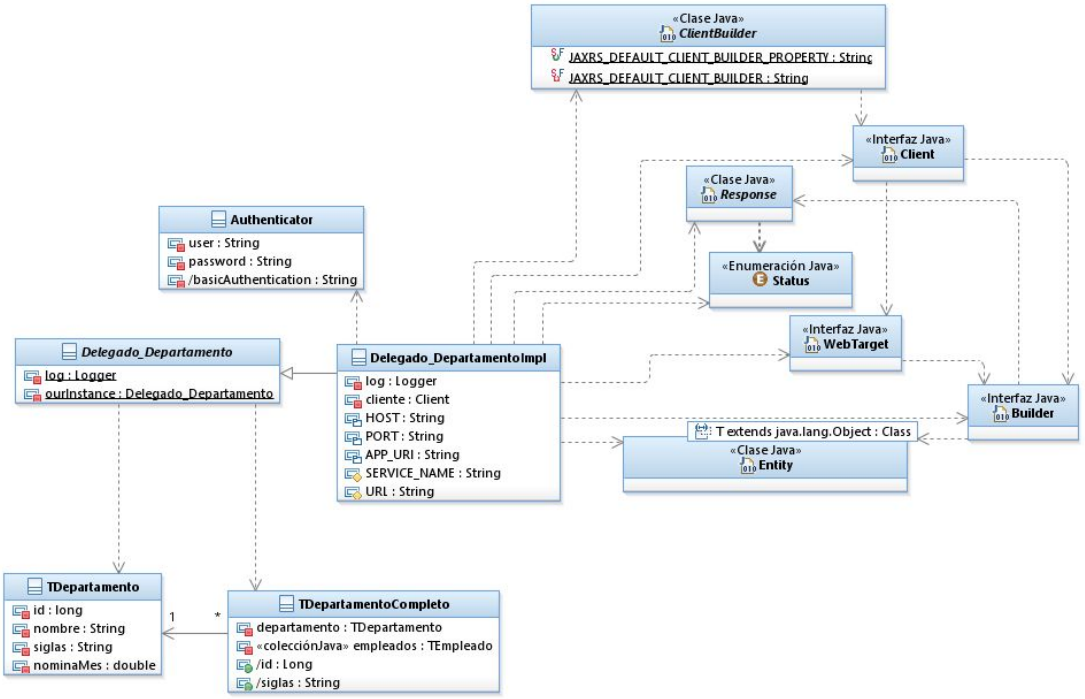

Figura 4.3.1. Diagrama de clase de Delegado del Negocio de Departamento

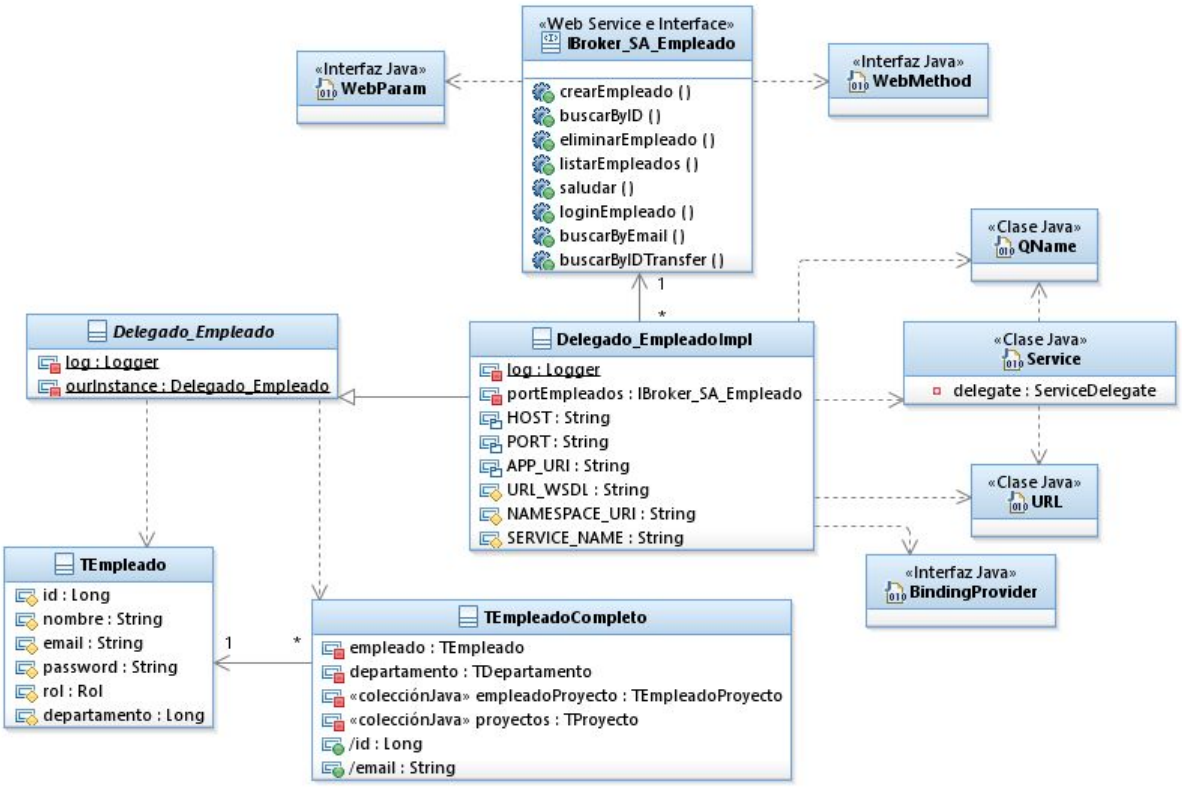

Figura 4.3.2. Diagrama de clase de Delegado del Negocio de Empleado

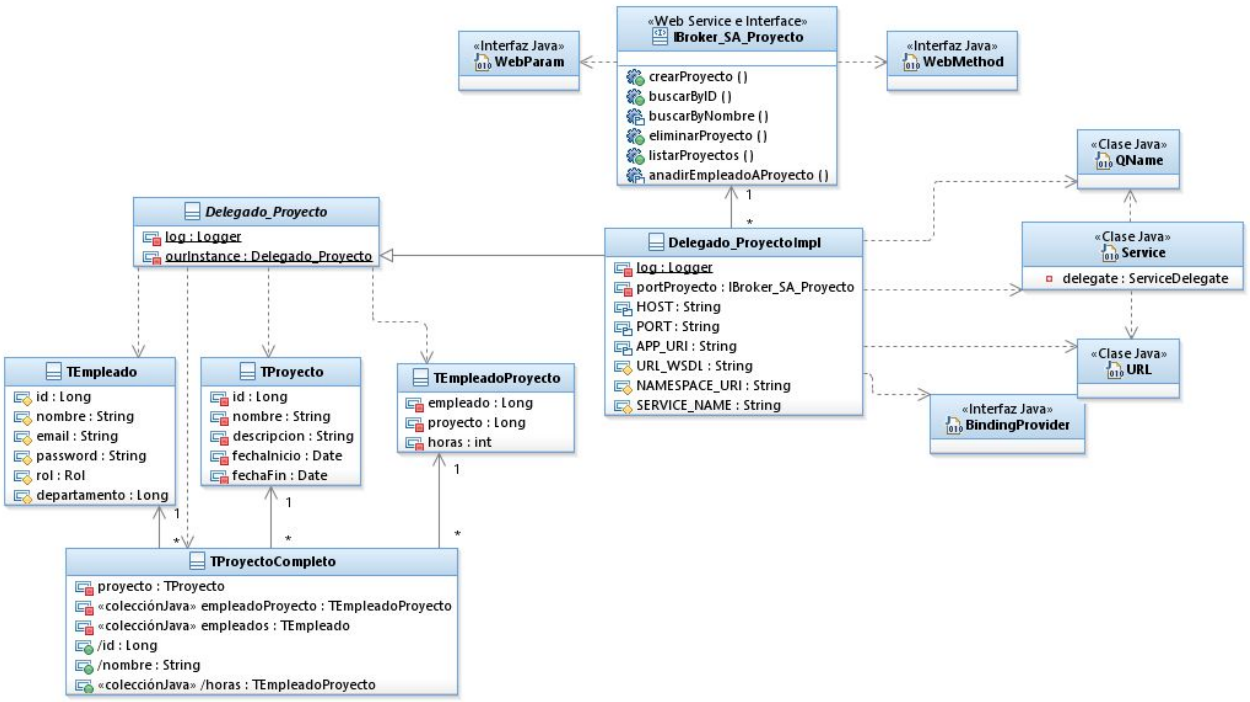

Figura 4.3.3. Diagrama de clase de Delegado del Negocio de Proyecto

La Figura 4.3.4 muestra la invocación del delegado del negocio de departamento al servicio web REST utilizando las clases JAX-RS.

La Figura 4.3.5 muestra la invocación del delegado del negocio de empleado al servicio web SOAP utilizando las clases JAX-WS.

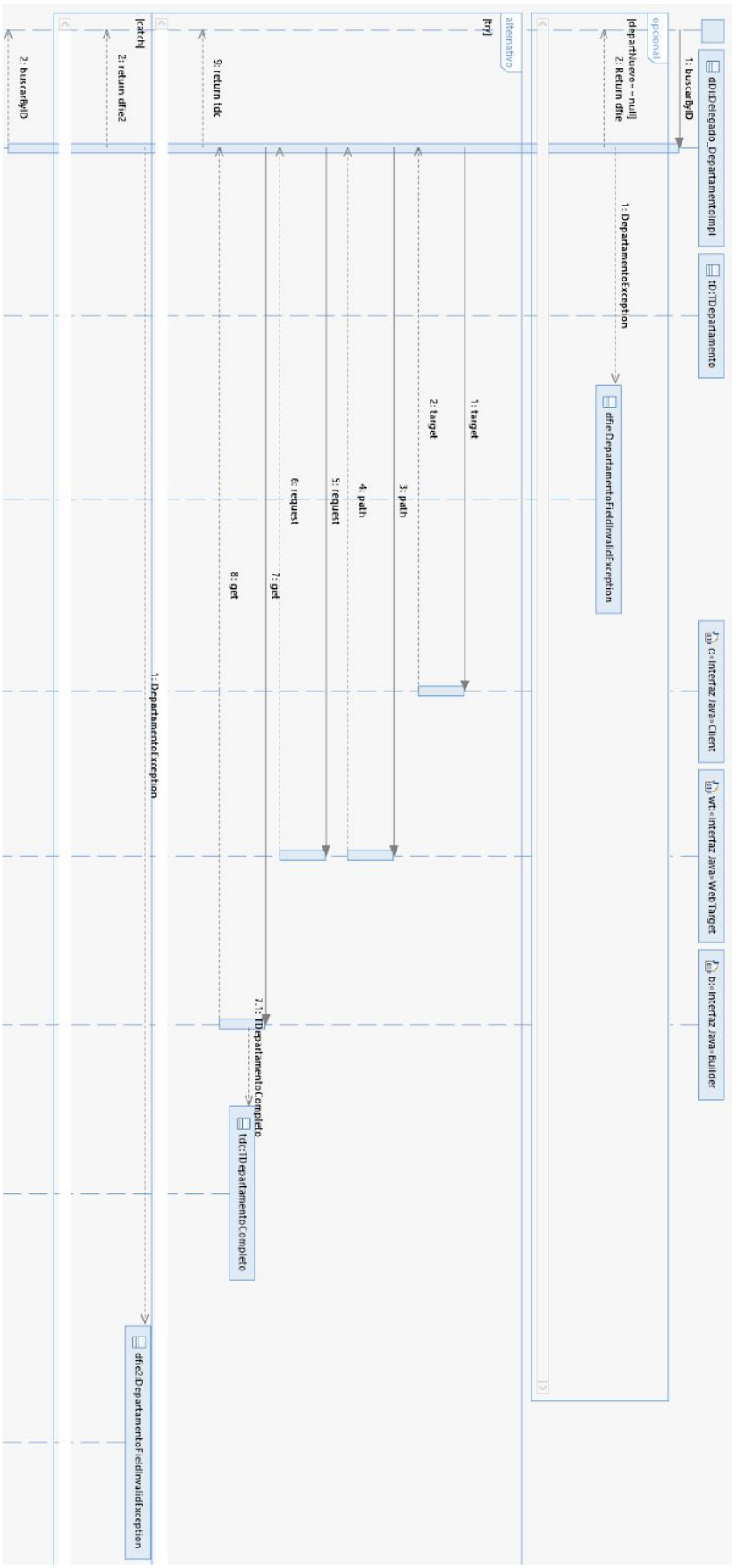

Figura 4.3.4. Diagrama de secuencia de buscar Departamento por ID en Delegado Departamento

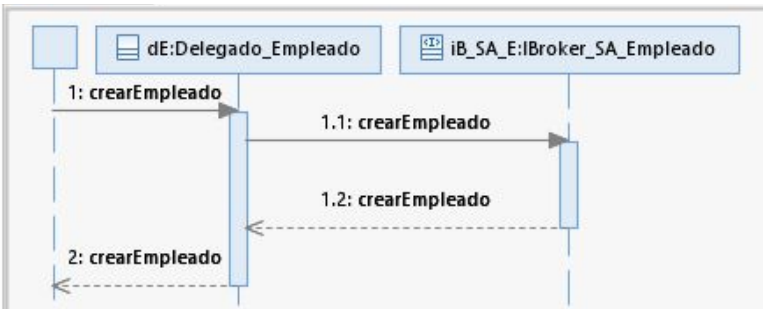

Figura 4.3.5. Diagrama de secuencia de Crear Empleado en Delegado Empleado

Como se puede ver en las figuras 4.3.4 y 4.3.5 la diferencia entre los Delegados REST y SOAP es abismal. Esto se debe a que JAX-WS provee un marco que abstrae completamente del protocolo HTTP, de forma que una vez creado el puerto de enlace con el servicio, el acceso a los métodos es trivial. De la misma forma el retorno de datos va implícito en la interfaz, al igual que las excepciones que pudieran propagarse desde la aplicación servidor, lo que hace que una vez configurado su uso sea igual al de una clase local a la aplicación.

Por otra parte, aunque el marco JAX-RS abstrae también del protocolo, éste obliga a construir la petición con el cliente, indicando el verbo, la URI, el tipo de datos que se va a enviar o recibir, y una vez que se realiza la petición el retorno de datos hay que controlarlo usando los códigos de estado HTTP que son devueltos por el servidor en función de cómo terminó la operación realizada. Esto obliga en cada aplicación a traducir las excepciones a códigos de estado y viceversa.

Capítulo 5. Capa de negocio de la aplicación servidor

5.1 Principales patrones utilizados e implementación con tecnologías de desarrollo

A continuación, se describe la capa de negocio de la aplicación servidor, es decir, el que implementa los servicios web. Los servicios web se exponen a través de *web service brokers*, WSBs, (Alur et al., 2003) invocados desde los delegados que conectan con la interfaz JSF y que exponen los servicios de aplicación (Alur et al. 2003) encargados de implementar todas las reglas de negocio del sistema. Así, los WSB exponen estos servicios de aplicación como servicios web SOAP y REST. Los WSBs usan una factoría abstracta (Gamma et al, 1994) para crear los servicios de aplicación, que abstrae a los clientes (el broker en este caso) de su implementación.

Para la transmisión de los datos entre la capa de negocio y la de presentación se usan objetos *transfer* (Alur et al., 2003). Estos son meros *POJOs* que contienen los atributos con los datos de las entidades del sistema y sus métodos *getter* y *setter*. Los servicios de aplicación reciben estos *transfers* (y los retornan con el resultado de la operación realizada) pero dentro los servicios lo convierten a las entidades JPA con las que trabaja. Se ha decidido utilizar *transfers* en vez de entidades JPA para externalizar los datos de los servicios web por dos razones: (i) evitan ciclos en grafos de objetos, lo que dificulta tremendamente la serialización de los datos; y (ii) evitan grafos *podados*, como, por ejemplo, el de un empleado vinculado a su departamento: el departamento sólo lleva un empleado, y no todos los que les correspondería a nivel dominio.

Si bien el primer problema se puede resolver utilizando librerías como MOXy (McLaughlin, 2002), la vinculación a una librería concreta, y la existencia del segundo problema, nos llevó a utilizar una solución basada en transfers para el envío de datos entre aplicaciones.

Para la capa de integración se ha utilizado JPA (implementado por Hibernate) como implementación del *almacén del dominio* (Alur et al., 2003), por lo que no ha sido necesario su programación. Su funcionamiento básico es el siguiente: cuando los servicios de aplicación requieren de algún objeto persistente, crean una `EntityManagerFactory` que a su vez le provee de un `EntityManager`. Con éste, se crea una transacción para controlar, entre otros, el acceso concurrente a la base de datos. Después, se realiza la operación y se cierra la transacción con un *commit* o un *rollback* en caso de haber un fallo. Los diagramas de secuencia incluidos en esta sección describen el proceso con mayor formalismo. Se ha decidido utilizar una gestión de la concurrencia *optimista* (Fowler, 2002), implementada con bloqueos optimistas y optimistas de incremento forzado JPA, según el caso (Keith et al, 2018).

Las siguientes figuras muestran los diagramas de clase que explican la estructura de las entidades JPA (Figura 5.1.1), los *transfers* (Figura 5.1.2), los WSB (figuras 5.1.3, 5.1.5 y 5.1.7) y servicios de aplicación/negocio (figuras 5.1.4, 5.1.6 y 5.1.8) de los tres módulos desarrollados. Como puede verse en las figuras, los servicios de aplicación siguen una estructura común para los tres módulos. El módulo departamento tiene un WSB REST utilizando clases JAX-RS y los módulos Empleado y Proyecto tienen WSB SOAP utilizando clases JAX-WS.

Las anotaciones JAX-RS fundamentales para definir servicios web REST son `@Path` para vincular URLs con invocaciones de clases; `@GET`, `@POST`, `@PUT` y `@DELETE` para vincular los métodos HTTP de invocación con métodos de clases; y `@Consumes`, `@Produces` para hacer la deserialización/serialización de los parámetros de entrada/salida.

La anotación JAX-WS fundamental para definir servicios web SOAP es `@WebService`, que identifica tanto al servicio web, como a su interfaz de punto final (Hansen, 2007) si existiese (como es el caso de este trabajo).

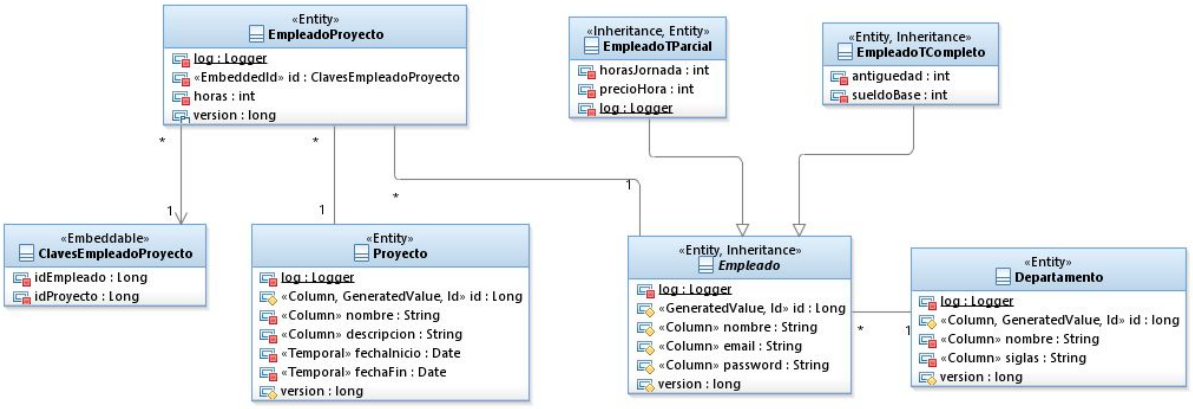

Figura 5.1.1. Diagrama de clases de las Entidades JPA

La relación entre los transfers tiene por partes principales: los *transfer* simples que representan un entidad de la base datos y los *transfer* completos que, acorde al patrón *transfer object assembler* (Alur et al., 2003), incluyen la información proveniente de relaciones de la entidad representada en el transfer, que pudiera ser necesaria para cada caso de uso concreto. (por ejemplo, los datos del departamento de un empleado concreto).

Los *transfer* simples son `TDepartamento`, `TProyecto`, `TEmpleado`, `TEmpleadoProyecto` y sus respectivas subclases, `TEmpleadoTParcial` y `TEmpleadoTCompleto`, que contienen únicamente los atributos correspondientes de las entidades (id, nombre, etc.) de la base datos.

Por otra parte, los *tranfers* completos son `TDepartamentoCompleto`, `TEmpleadoCompleto`, `TProyectoCompleto`. Por ejemplo, el *transfer*

`TEmpleadoCompleto` es el más complejo, contiene una subclase de `TEmpleado`, un `TDepartamento` para la relación 1 a N y dos colecciones de `TProyecto` y `TEmpleadoProyecto` para la relación N a M.

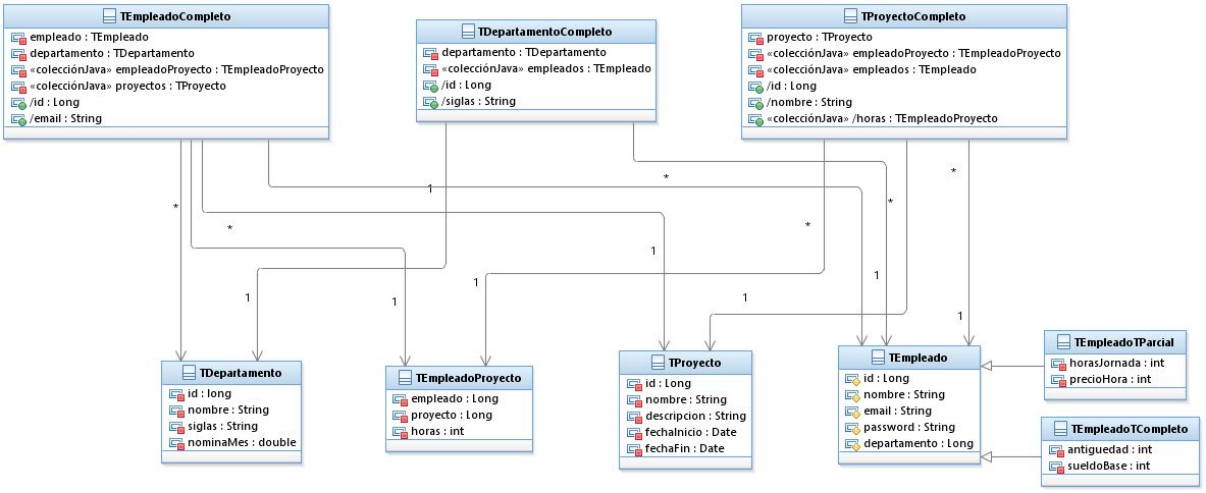

Figura 5.1.2. Diagrama de clases de los Transfers

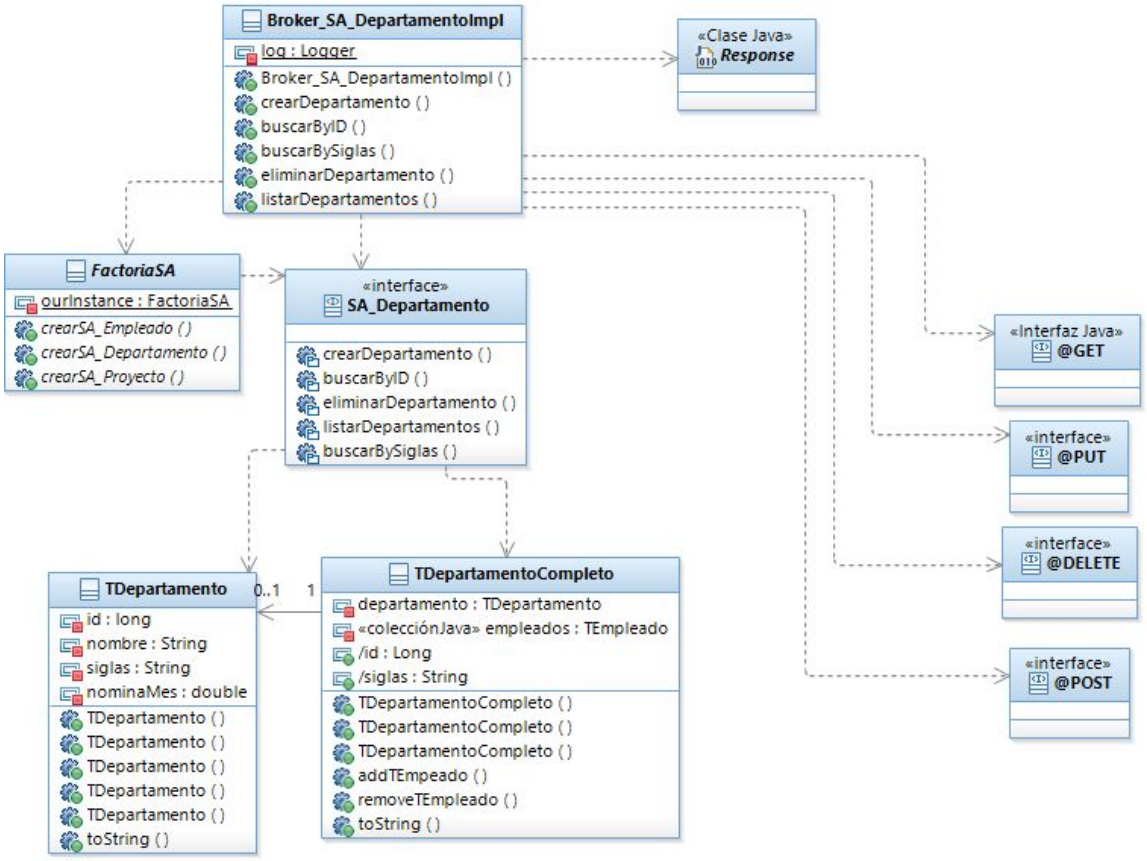

Figura 5.1.3. Diagrama de clases del web service broker de Departamento

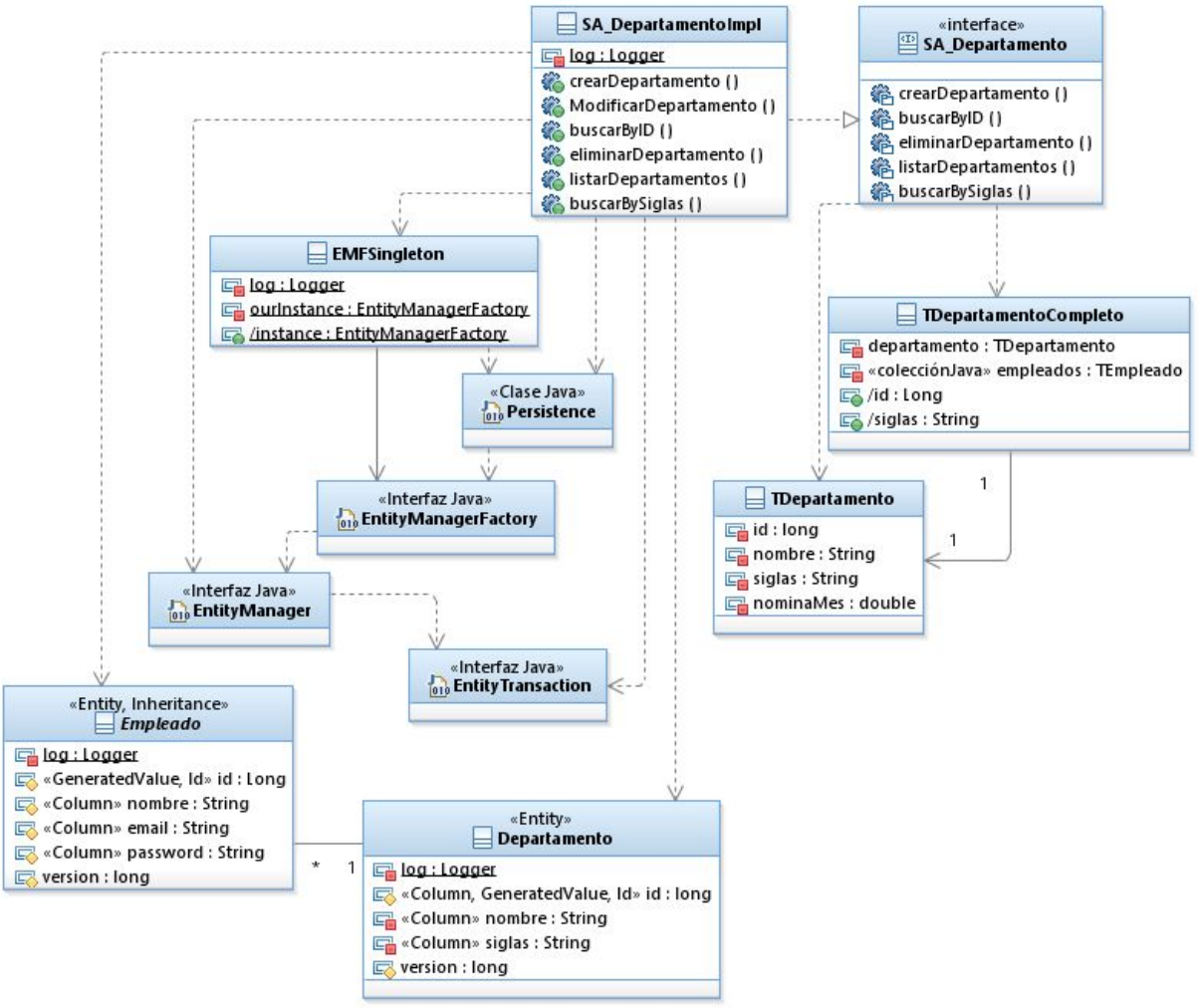

Figura 5.1.4. Diagrama de clases negocio para Departamento.

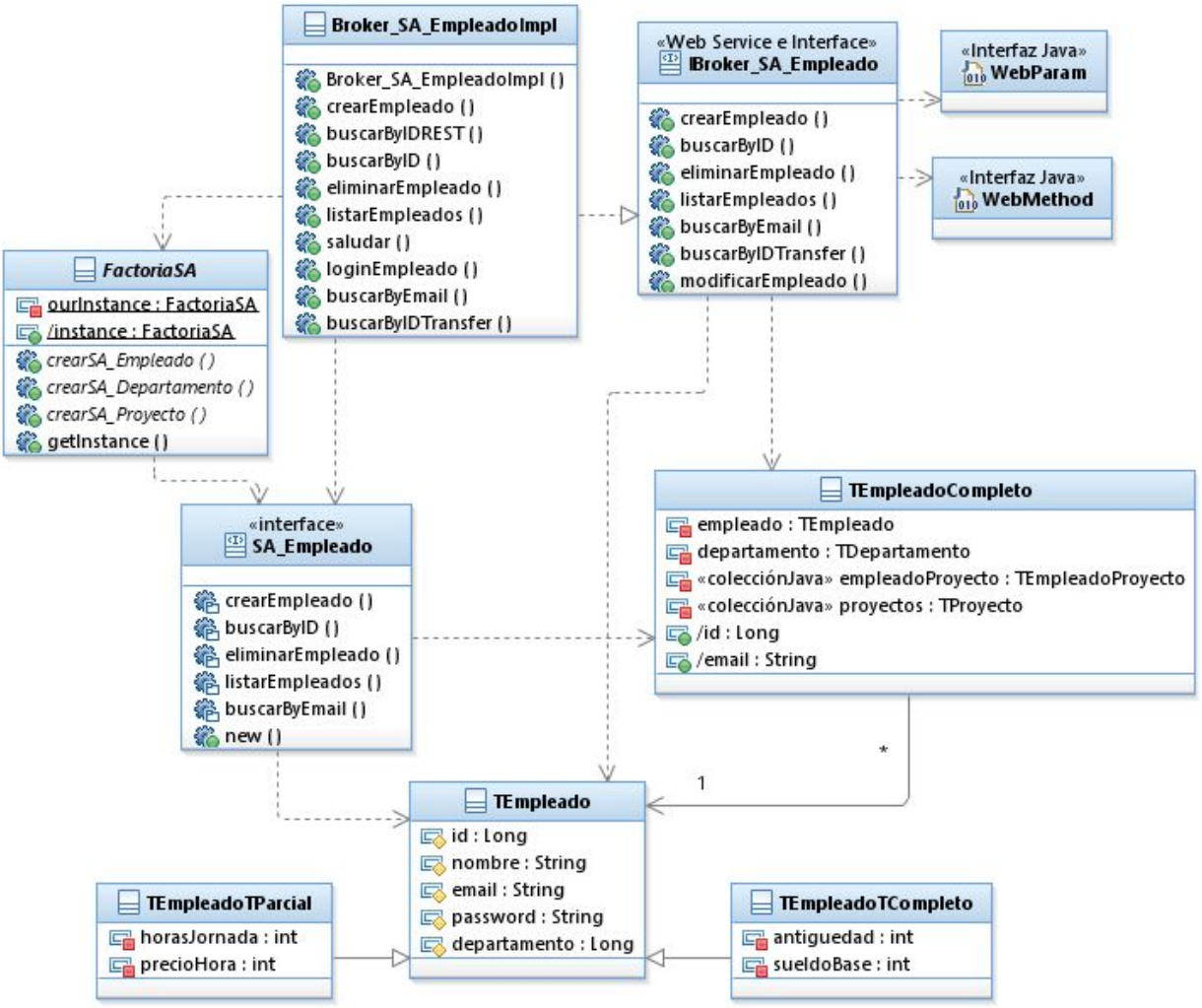

Figura 5.1.5. Diagrama de clases del web service broker de Empleado

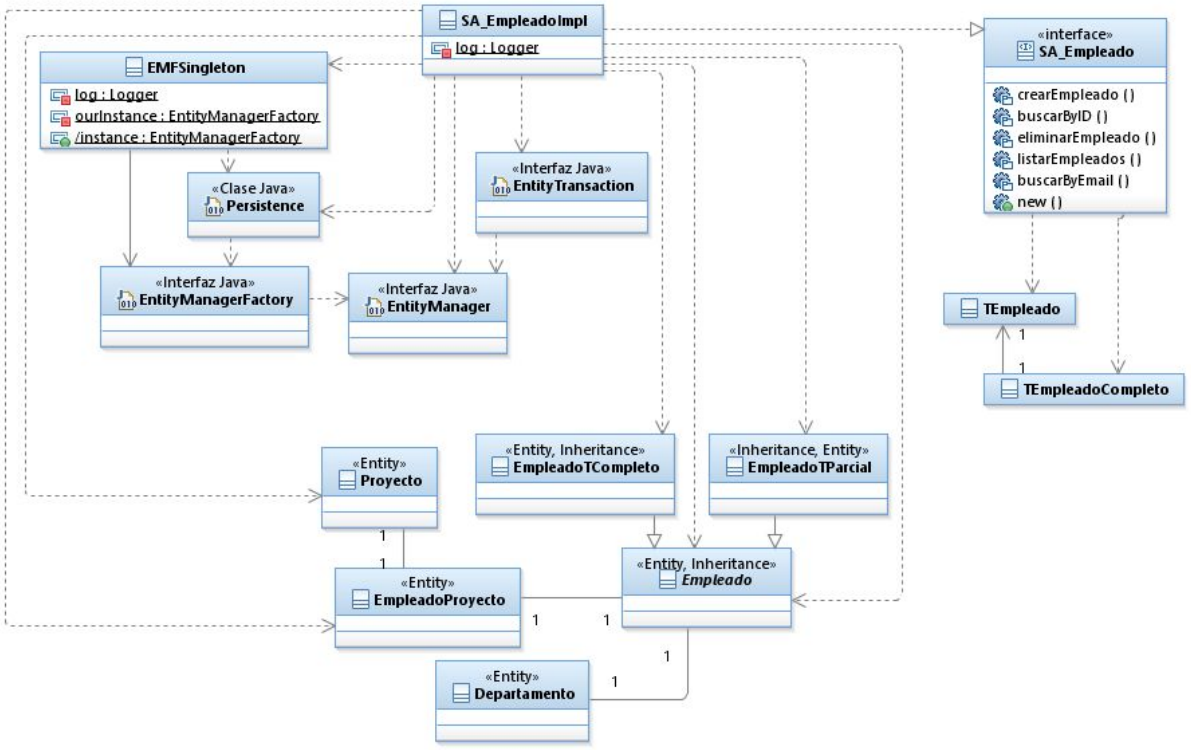

Figura 5.1.6. Diagrama de clases de negocio para Empleado

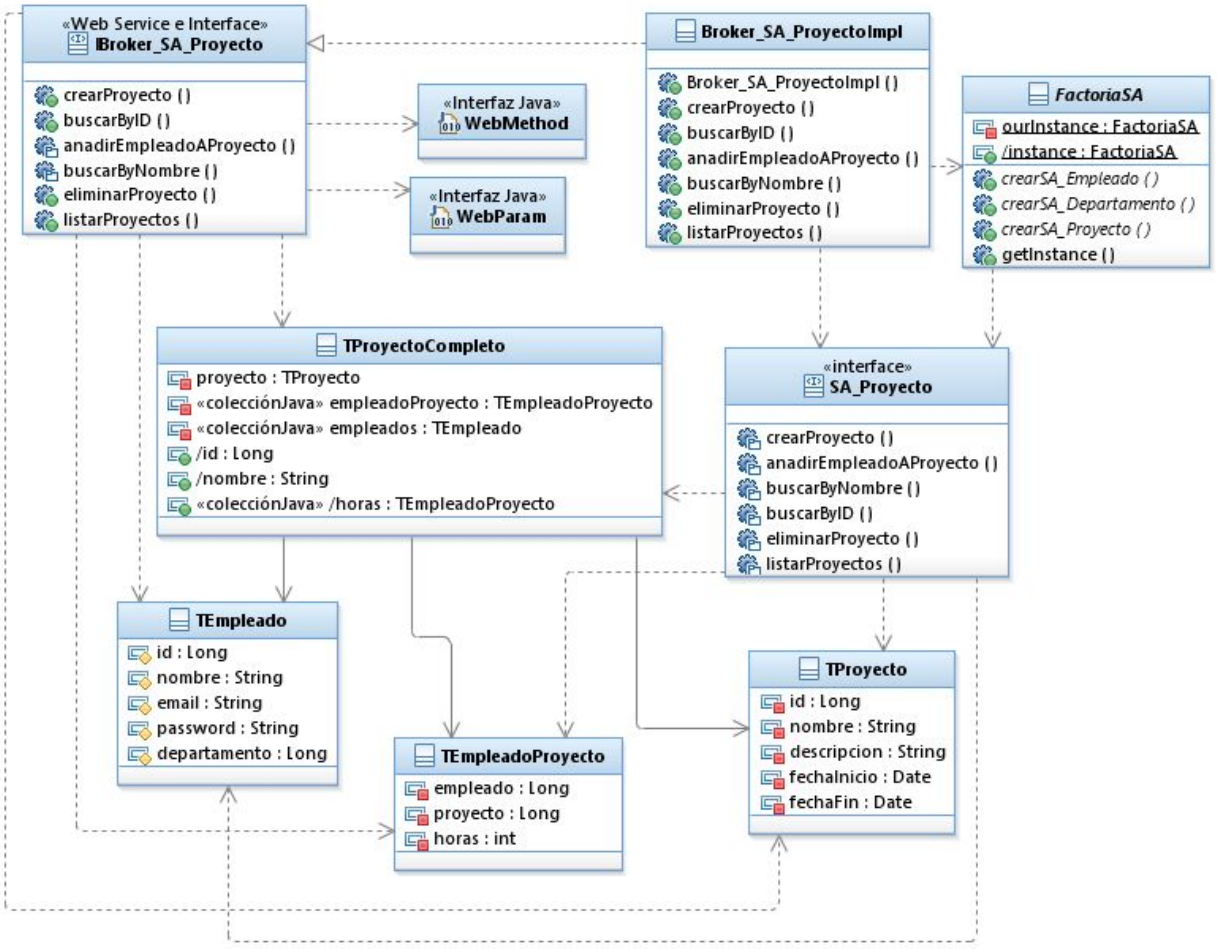

Figura 5.1.7. Diagrama de clases del web service broker de Proyecto

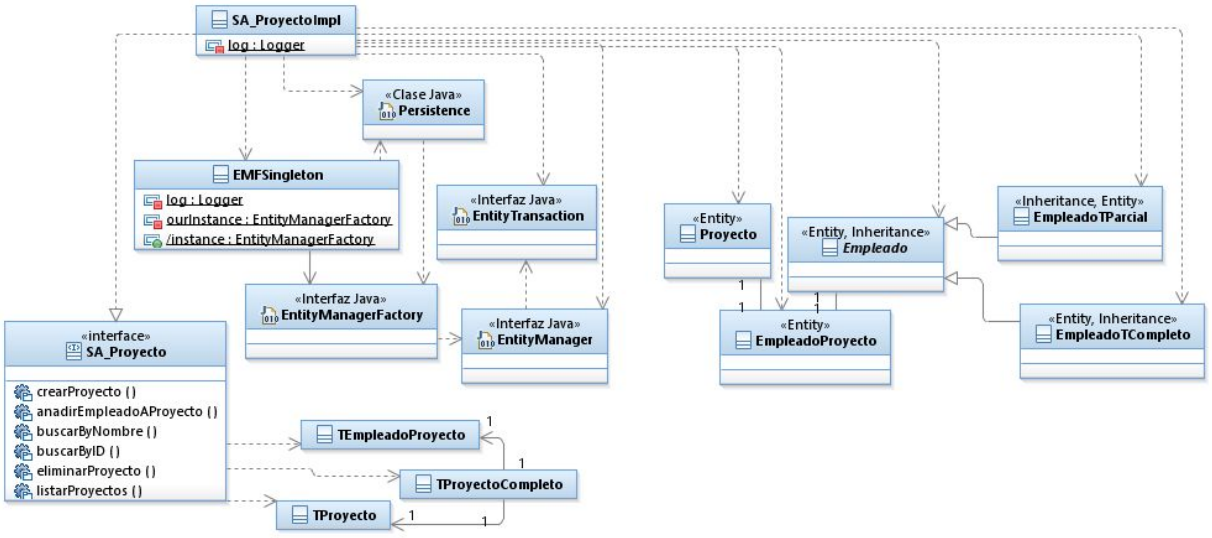

Figura 5.1.8. Diagrama de clases de negocio de Proyecto

Una vez definida la estructura de clases de la capa de negocio se puede pasar a los diagramas de secuencia. Estos diagramas muestran de forma detallada el comportamiento de los métodos implementados en el proyecto. Se han elegido los dos mismos requisitos de la capa de presentación para seguir una traza completa desde el usuario hasta la base de datos: *Buscar Departamento por ID* y *Crear un Empleado. De esta forma se muestra* cómo funciona la capa de negocio, y como se exponen los servicios mediante JAX-RS y JAX-WS respectivamente. También se puede ver cómo se conecta esta capa con la de integración, representada por JPA. Este comportamiento se puede extrapolar al resto de requisitos del sistema.

Con respecto a la gestión de la concurrencia, la Figura 5.1.10 muestra *buscar departamento por id*, incluida en la operación el *cálculo de la nómina de un departamento*. Se puede ver cómo el servicio de aplicación bloquea de forma optimista tanto al departamento, como a sus empleados, lo que previene de cambios en estos durante el cálculo de la nómina. Además, para evitar el problema de la inclusión de nuevos empleados en el departamento durante el cálculo de la nómina la Figura 5.1.12 muestra como el servicio de aplicación de empleado, bloquea al departamento con un bloqueo optimista de incremento forzado. Así, si el empleado se inserta antes del fin del cálculo de la nómina, el bloqueo del departamento durante el cálculo de su nómina, produciría el fallo de la transacción al hacer el commit, tal y como se desea. Tanto en este caso de uso, como en el resto, se ha utilizado una gestión de la concurrencia optimista, mucho más flexible que la pesimista (Fowler, 2009), implementada con las facilidades que proporciona JPA (Keith et al, 2018).

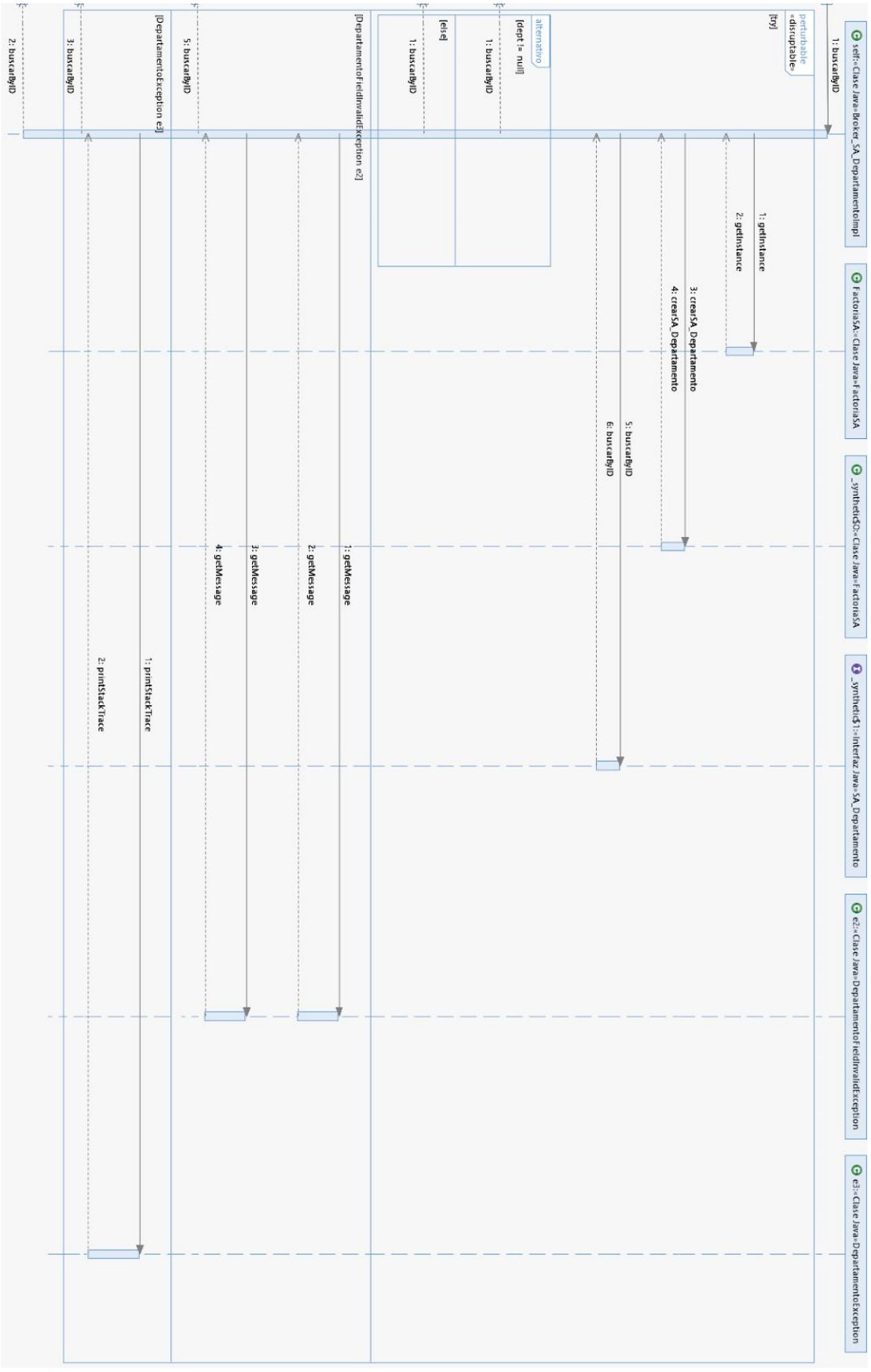

Figura 5.1.9. Diagrama de secuencia de buscar departamento por ID en en WSB de Departamento

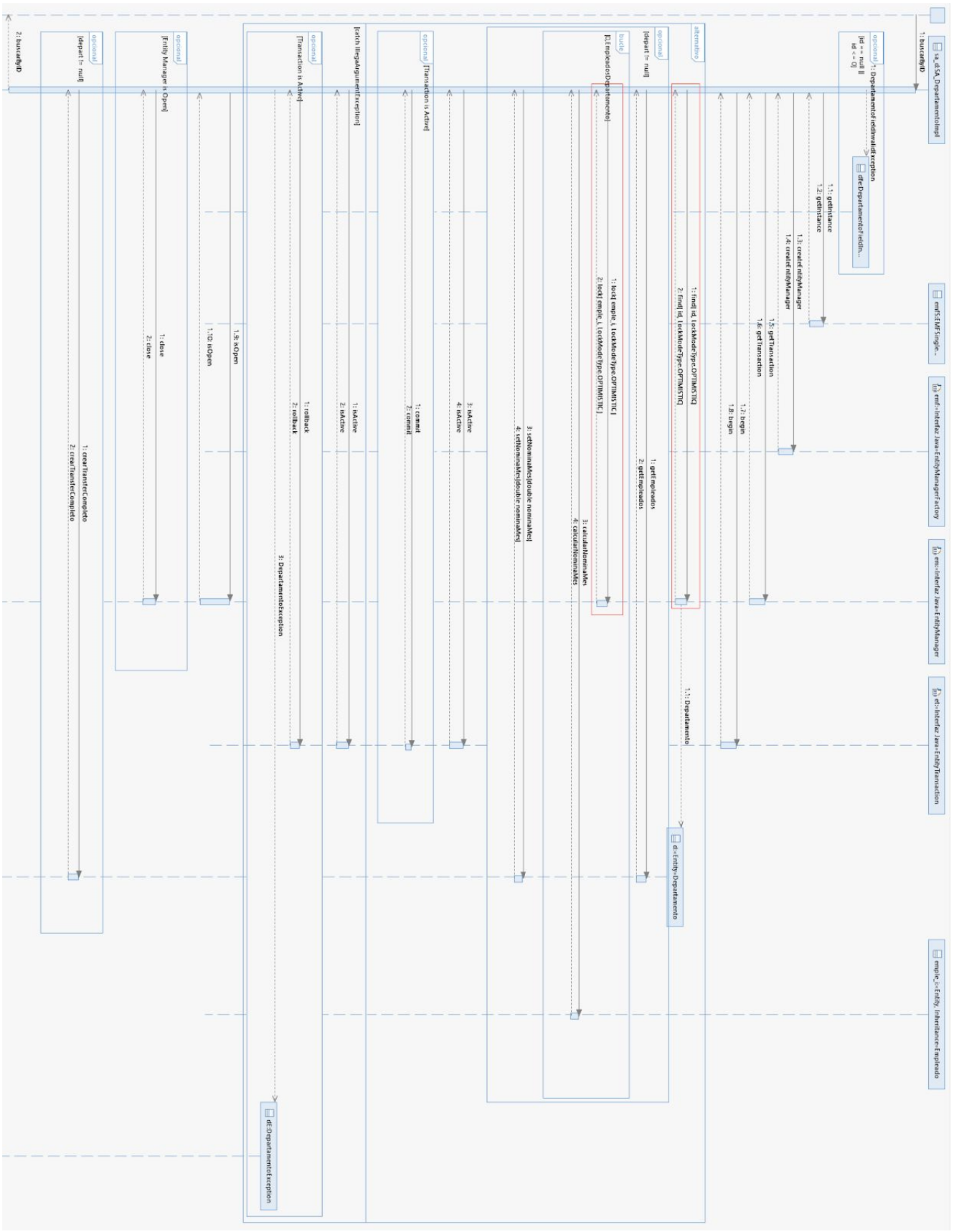

Figura 5.1.10. Diagrama secuencia de Buscar Departamento por ID en Servicio Aplicación Departamento

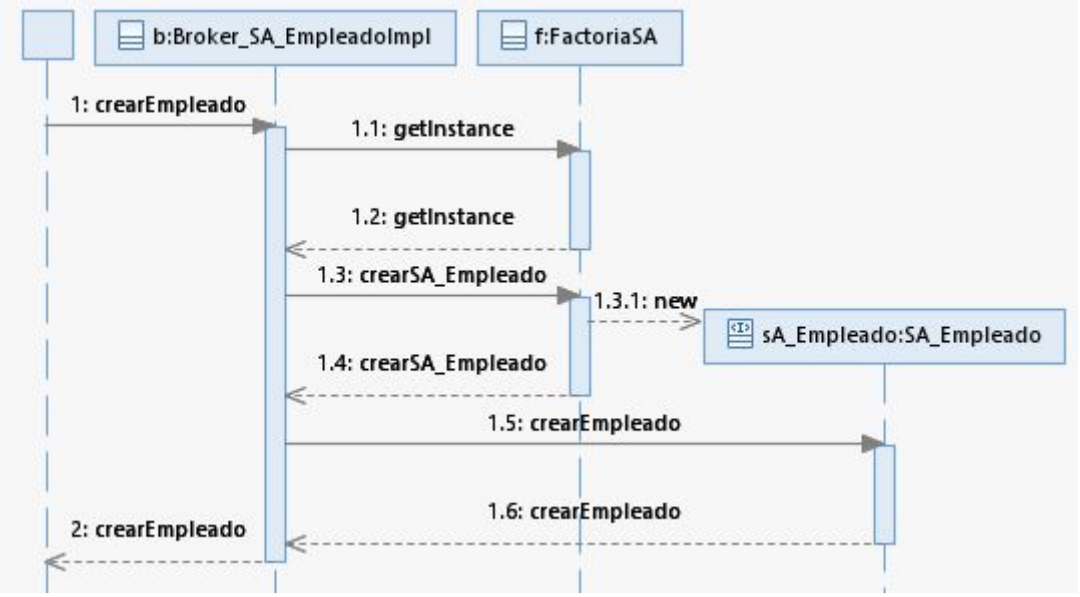

Figura 5.1.11. Diagrama de secuencia de crear empleado en el WSB de Empleado

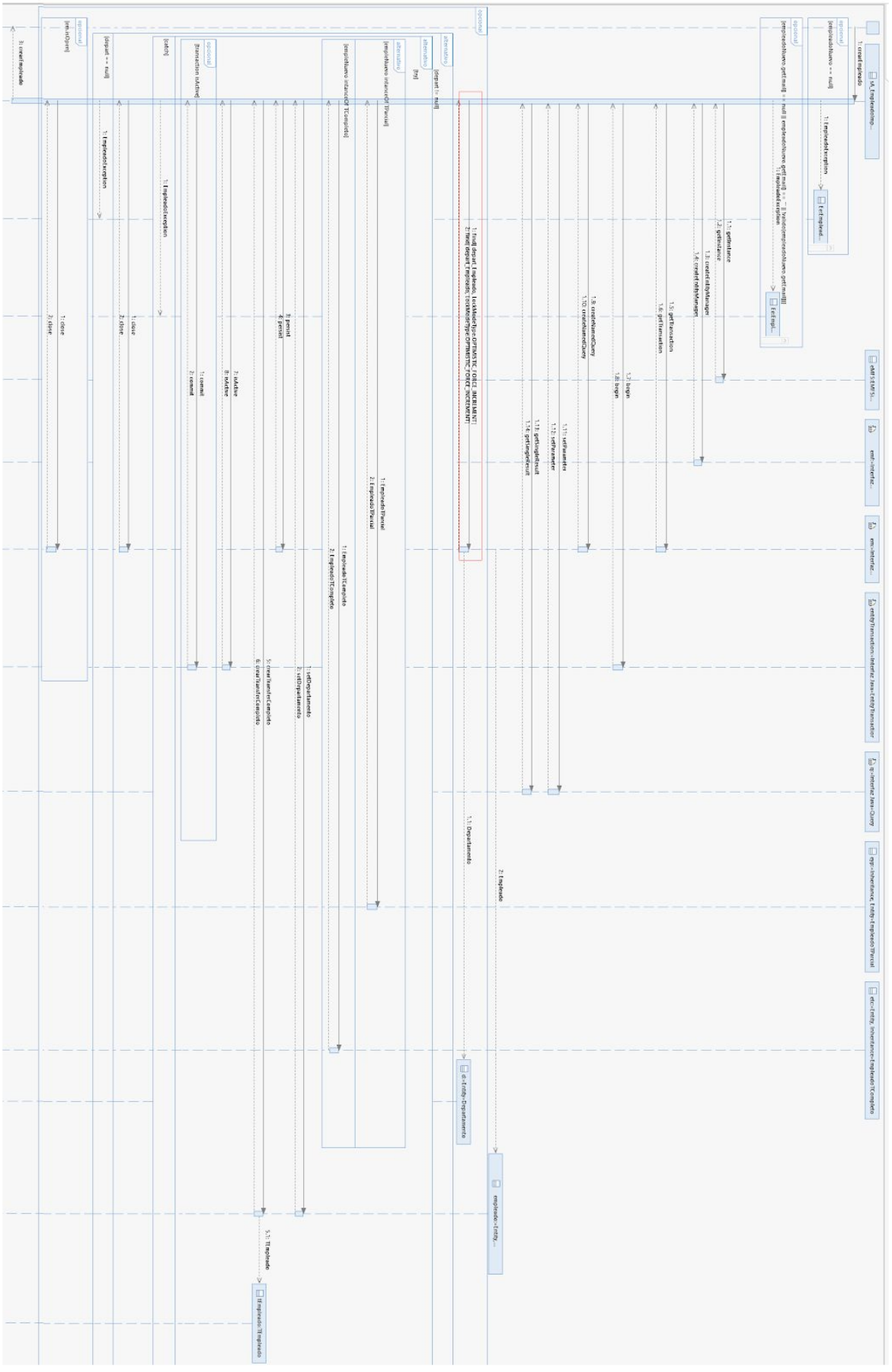

Figura 5.1.12. Diagrama secuencia de Crear Empleado en Servicio Aplicación Empleado

Capítulo 6. Prueba y Control de Versiones

6.1 Pruebas unitarias

Las pruebas de calidad se han realizado en dos partes:

- Las vistas HTML han sido probadas sin marcos de automatización, probando que funcionan bien los distintos formularios, que están bien conectados tanto con los métodos de los *Managed bean* como todos los campos de los formularios conectados con los atributos de los beans. Del mismo modo se han probado los enlaces de las páginas comprobando que redirigen bien, o se hace correctamente el login en el sistema, por ejemplo.

- En el resto de paquetes, se ha creado para ambas aplicaciones, una clase de test JUnit 5 por cada clase de código, siguiendo la misma jerarquía de paquetes que en el paquete principal. Se pueden encontrar todos los test en el directorio "`src/test/java`", especificado para proyectos Maven.

Grosso modo cada clase de test sigue una estructura de nombre concreta, el nombre de la clase que prueba terminada en "Test", por ejemplo, `DepartamentoBeanTest.java`.

Dentro de cada clase hay tres secciones destacables:

- Los atributos que requieren las pruebas de código, por ejemplo, la clase que se va a testar, transfers, etc y un `Logger` para las trazas de ejecución.

- Métodos de iniciación y finalización del entorno usando las anotaciones `@BeforeAll` para iniciar la clase a probar, y los transfer y relaciones entre ellos que sean necesarias para la ejecución de las pruebas. Y luego dos métodos con las anotaciones `@BeforeEach` y `@AfterEach` para crear algún objeto concreto en cada test y su eliminación.

- Métodos de Test: para cada prueba realizada se ha creado un método concreto. De esta forma se obtienen pruebas muy atómicas que permiten encontrar y solucionar fácilmente los fallos encontrados. Por ejemplo, para los métodos de creación se ha testeado casos en los que en parámetros sea `null`, un objeto vacío, el id exista en el sistema, número fuera de rango, etc.; para los de búsqueda de información, por ejemplo, se han probado que el parámetro sea `null`, un atributo principal de búsqueda (como pueda ser ID o email) sean vacíos o nulos, no existan, etc.

6.2 Control de versiones

Para el control de versiones se ha usado GitHub. El procedimiento, explicado en el apartado 2.3.2 GitHub y 2.3.3 Jenkins, ha consistido en hacer una subida de código al repositorio (con un comentario de los cambios y adiciones realizadas) tras cada “jornada” de trabajo.

El repositorio se divide en dos ramas, una Master con el código que ha pasado todas las pruebas y la Development sobre las que se ha ido desarrollando el código.

El código de la aplicación cliente y servidor está accesible en mi cuenta personal de GitHub:

- Cliente: https://github.com/hunzaGit/TFG_cliente
- Servidor: https://github.com/hunzaGit/TFG_server

Capítulo 7. Conclusiones y trabajo futuro / Conclusions and future work

Versión española

7.1 Conclusiones

El desarrollo de una aplicación empresarial, incluyendo seguridad, atravesando todas las capas de la misma es una cuestión extremadamente compleja.

En el GIS se proporcionan conocimientos para crear una aplicación multicapa, transaccional y que esté preparada para resolver el acceso concurrente a los datos. Sin embargo, el desarrollo de una aplicación de extremo a extremo, utilizando una plataforma de desarrollo e integración continua es algo bastante complejo. Así, considerando todos los marcos y tecnologías de desarrollo, de los once utilizados en este trabajo, sólo tres se habían utilizado en el grado: JPA (Hibernate), SGBDR (MariaDB) y SCV (GitHub).

El uso de tantos marcos y tecnologías nuevas para mi en la elaboración de este TFG ha sido un verdadero reto, que ha merecido la pena, ya que tras finalizarlo tengo la sensación de haber completado mi formación y, quizá, de haber subido un escalón de capacitación profesional en mi perfil futuro como Ingeniero de Software.

Algunos marcos y herramientas eran robustas y cuentan con el *paraguas* de grandes organizaciones como Oracle (por ejemplo, JPA) o Apache (por ejemplo, Tomcat). Otras, sin embargo, eran mucho más novedosas y punteras (por ejemplo, Jenkins).

Sin embargo, el principal reto no ha sido utilizar una única tecnología, sino hacer funcionar todas de forma coordinada, integrando su uso de capa a capa. De esta forma, guiados por los patrones de diseño de la arquitectura multicapa, los marcos encajan como un puzle, formando un ecosistema de desarrollo. La dificultad está en conseguir que funcionen integradas, lo que no es simple debido a las incompatibilidades entre librerías, incompatibilidades que se detectan después de horas de desarrollo infructuoso.

Esto ha conllevado un esfuerzo de desarrollo muy alto, tanto para conocer el funcionamiento detallado de marcos y herramientas, como para navegar por trazas de error, buscando soluciones y experiencias de uso en foros específicos de desarrollo por Internet.

Cabe destacar aquí la doble naturaleza de un proyecto software industrial: arquitectónica (guiada por un catálogo bien definido de patrones de diseño) y de implementación (apoyada en marcos concretos). Desde nuestro punto de vista, la fundamental es la arquitectónica, pero una aplicación real necesita marcos concretos de implementación. Así, este proyecto se centra en los marcos de la plataforma J2EE, pero su arquitectura es extensible a otros

marcos de esta o de otra plataforma, marcos, de una forma u otra, de uso similar a los aquí utilizados.

No es la primera vez que me enfrento a un proyecto web, pero con anterioridad lo había realizado con JavaScript (Duckett, 2014) en una aplicación *End-to-End* (Mikowski, 2013) (JavaScript de extremo a extremo, de cliente a BBDD, usando Node.js (Brown, 2014) en servidor y MongoDB (Membrey & Hows & Plugge, 2014) como BBDD). Comparado con JSF, aunque son paradigmas completamente distintos a la hora de afrontar el desarrollo de la aplicación cliente, la diferencia, especialmente en la parte de cliente, es notoria tanto en la forma de diseñar como en la de programar. Con JavaScript todo el cliente se ejecuta en el navegador y realiza peticiones REST al servidor, lo que unifica mucha lógica en cliente, mejora mucho los diseños web e interfaces gráficas, pero, por esa misma razón, ralentiza un poco su desarrollo comparado con la rapidez de JSF por la definición declarativa de las normas de navegación.

Cabe destacar la potencia de uso de JAAS, que permite implementar una autorización y autenticación para el acceso a páginas web de manera casi declarativa. Por tanto, el uso de este marco hace más sencillos los procesos de autorización y autenticación que en una implementación manual de los mismos.

Con respecto a la invocación segura de los servicios web, delegarlo en la capa de transporte a través del protocolo HTTPS lo simplifica bastante, garantizando así un transporte seguro de punto a punto. Las ventajas son sencillez y velocidad, y el gran inconveniente es la falta de seguridad extremo a extremo.

La persistencia de datos se ve muy facilitada en Java con el marco JPA, y en este caso concreto, su implementación Hibernate. JPA facilita el acceso a los datos en formato de objetos, su carga dinámica, la gestión de transaccional y la gestión de la concurrencia.

Respecto a los dos tipos de servicios web implementados en este proyecto, una vez acabado, cabe destacar la sencillez de uso en Java de servicios web SOAP y REST. Los servicios REST ya los había utilizado anteriormente en aplicaciones JavaScript, por lo que conocía su ligereza y facilidad de implementación y despliegue. En este proyecto he utilizado por primera vez los servicios web SOAP, y gracias a JAX-WS, su programación y despliegue es aún más sencillo que los servicios web REST. Además, en una aplicación empresarial con miles de líneas de código, el uso de excepciones, que permiten los servicios SOAP a través del interfaz WSDL, facilitan su uso y depuración.

Por otro lado, el uso de un marco de pruebas unitarias como JUnit, facilita enormemente el desarrollo de código de calidad desde el inicio. Aunque es cierto que supone una gran cantidad de esfuerzo para pensar y codificar todas las pruebas de cada método programado, una vez acostumbrado, no se concibe un método de código funcional sin sus correspondientes pruebas. Por otro, reduce el estrés al programador, cuando se va a ejecutar el código, por miedo a que falle.

Por último, el entorno de integración continua, al que dediqué un especial interés por la fascinación personal que me produce la automatización de tareas y procesos, me ha parecido una herramienta potentísima para la gestión de proyectos a mayor escala. Entiendo de primera mano que cada vez más compañías incorporen esta herramienta a su *stack* tecnológico. Aunque de forma muy sencilla, he podido comprobar cómo se construye el proyecto, se lanzan las pruebas, se mantienen diferentes versiones de código en el repositorio en función de su estabilidad, se informa de los fallos al programador, e incluso se despliega en Apache Tomcat, todo ello de forma automática, una vez configurado el entorno.

En cuanto al código funcional para la aplicación cliente se han realizado 11 clases de test con un total de 113 test unitarios y una cobertura de código del 92% de las clases (39/42), un 60% de los métodos (162/271) y un 62% de las líneas de código (560/894).

Esta aplicación consta de un total de 17538 líneas de código (excluyendo los css de la librería de Bootstrap), de las cuales:

- 7636 líneas de Java, el 57% de código ejecutable, el 10% de comentarios y el 33% de líneas en blanco (usadas en su mayoría para dar mayor legibilidad al código).

- 3020 líneas de XHTML, 77% de código ejecutable, el 1% de líneas comentadas y el 22% de líneas en blanco (usadas en su mayoría para dar mayor legibilidad al código).

Para la aplicación servidor se han realizado 10 clases de test con un total de 190 test unitarios y una cobertura de código del 94% de las clases (37/39), un 54% de los métodos (204/375) y un 60% de las líneas de código (986/1595).

Esta aplicación consta de un total de 10654 líneas de código, de las cuales:

- 8498 líneas de Java, el 58% de código ejecutable, el 9% de comentarios y el 34% de líneas en blanco (usadas en su mayoría para dar mayor legibilidad al código).

A pesar de todo este esfuerzo, el enfrentamiento con cada marco me ha permitido aprender cosas nuevas, a valorar mucho más el trabajo de un Ingeniero de Software diseñando una arquitectura software, e incluso el trabajo de un Ingeniero de Sistemas, centrado en el despliegue y funcionamiento de aplicaciones empresariales. Es por ello que el esfuerzo se ha visto recompensado por la superación de nuevos retos.

También y, para mi no menos importante, he de destacar la madurez y resiliencia adquirida para afrontar problemas y encontrar soluciones a largo plazo, distintos —la mayoría— y superiores —todos—, a los planteados en la carrera.

7.2 Trabajo futuro

A pesar de haber afrontado muchos retos de desarrollo, hay algunos marcos y herramientas, que me hubiera gustado utilizar en este TFG, pero no fue posible debido a la necesaria administración de tiempos y esfuerzos.

En el apartado de seguridad, si bien es cierto que se ha usado un mecanismo de autenticación y autorización para la invocación de servicios web SOAP y REST basado en la capa de transporte, me hubiera gustado utilizar WS-Security para servicios web SOAP. Actualmente, en este trabajo, la seguridad en la invocación de servicios web se delega en el protocolo HTTPS y en el contenedor (Tomcat). Esta es una solución robusta y que no sobrecarga la aplicación. Sin embargo, no proporciona seguridad de extremo a extremo, sino punto a punto. WS-Security, sí que proporciona seguridad extremo a extremo, encriptando el mensaje WSDL. Sin embargo, el sobrecoste computacional de su uso es bastante superior al del protocolo HTTPs.

En el apartado transaccional, no se ha abordado el problema de las transacciones distribuidas, ya que las transacciones de esta aplicación se aplicaban sobre un único recurso transaccional (MariaDB). Java Transaction API, JTA, (Little et al, 2004) resuelve este problema. Sin embargo, se ha optado por no utilizarlo en este TFG.

Las pruebas de carga sobre aplicaciones web son un elemento fundamental para el despliegue y uso de una aplicación web. JMeter (Halili, 2008) permite realizar y automatizar este tipo de pruebas, pero no ha sido utilizado en este trabajo.

Finalmente, el catálogo de patrones de seguridad multicapa contiene muchos más patrones que los aquí utilizados. La inclusión de todos los patrones de seguridad multicapa resultaría muy interesante, pero eso, en sí mismo, constituirá, en volumen de trabajo, otro TFG nuevo.

English version

7.1 Conclusions

The development of an enterprise application that includes security, going through all its layers, is a complex issue.

The degree in software engineering gives knowledge to build a transactional multitier application that support concurrent access to data. However, the end-to-end development of enterprise applications using continuous integration tools is a quite complex issue. Thus, taking into account all the frameworks and technologies used in this development, only three of eleven (JPA, RDBMS, and CVS) had been used in the degree.

The use of this amount of new frameworks and technologies has been a challenge, which has been well worth, and that has improved my training, making me to get another step up the ladder as software engineer.

Some frameworks and technologies are consolidate and have the support of key organizations such as Oracle (e.g. JPA) or Apache (e.g. Tomcat), while others are newer such as Jenkins.

However, to use a single framework was not the main challenge. The coordinated use of all of them was the main challenge. Thus, guided by the multitier design patterns, the frameworks fit like the pieces of a puzzle, making a development ecosystem. The main difficulty is to make them to work simultaneously. This has implied a huge development effort.

Is important to remark the double nature of enterprise software projects: architectural (guided by a well-defined design pattern catalogue) and implementation-focused (supported by specific frameworks). From my point of view, architecture is more important than implementation, but real applications need real implementations. Thus, this work uses J2EE as implementation technology, but its architecture can be used with other frameworks belonging to this or another platform.

This is not my first web project, because I had made a JavaScript project (Duckett, 2014) in an *End-to-End* application (Mikowski, 2013, using Node.js (Brown, 2014) as server, and MongoDB (Membrey & Hows & Plugge, 2014) as DBMS. Compared with JSF, JavaScript has not the declarative power when defining declarative navigation rules in the presentation tier.

It is important to remark the power of JAAS, which allow us to implement authentication and authorization of users for accessing to web pages in a declarative way. Therefore, the use of this framework make easier these issues than their manual implementation.

Regarding secure invocation of web services, delegating them in the transport tier, using HTTPS protocol make it easier, providing a secure point-to-point transport. Simplicity and speed are the main advantages, but lack of extreme-to-extreme security is the main drawback.

JPA simplifies data persistence, providing also dynamic load, transactional management, and support for their concurrent access.

Regarding SOAP and REST services used in this project, Java frameworks simplifies their implementation and use. REST services are simple to use and deploy (I had used them in the previous JavaScript web application), but JAX-WS make very simple the use of SOAP services, even simpler than REST services. In addition, SOAP services enable the use of exceptions, which make their development easier.

The use of testing frameworks such as JUnit make easier the development of quality code from the beginning. The use of these frameworks increase the initial effort of development, but is well worth because help to make quality software, that provides confidence to developers.

Finally, the continuous integration tool is a very useful tool for enterprise projects. After using it, I understand the reason why is used in enterprise development.

Regarding the code of the client application, I have made 11 test classes with 113 unitary tests, which cover 92% of classes (39/42), 60% of methods (166/271) and 62% of lines of code (560/894).

The client application has 17,538 lines of code (excluding Bootstrap CSSs):

- 7,636 lines of Java code: 57% code, 10% commentaries, and 33% blank lines for code legibility.
- 3,020 lines of HTML, 77% reusable code, 1% commentaries and 22% blank lines for code legibility.

The application server has 10 test classes with 190 unitary test that cover 94% of classes (37/39), 54% of methods (204/375) and 60% of lines of code (986/1595).

The server application has 10,654 lines of code. 8,498 lines of Java code: 58% code, 9% commentaries, and 34% blank lines for code legibility.

In spite of this effort, to face each framework has taught me: new concepts, to value the work made by software engineers designing software architectures, and even, the work of system engineers focused on the development of enterprise applications. Therefore, the effort has been rewarded by overcoming new challenges.

Finally, it is very important the maturity and resilience achieved during the development of this work, seeking long term solutions, most of them, more complex than those faced during the degree.

7.2 Future work

In spite of the effort made in this work, important frameworks and technologies have not been used.

The secure invocation of web services has been supported by the transport tier (HTTPS), and the web container (Tomcat). However, the use of WS-Security provides an extreme-to-extreme security encrypting SOAP messages. This is a powerful solution that overloads the hardware infrastructure.

Regarding transactional management, distributed transactions have not been used in this work. Java Transaction API, JTA, (Little et al, 2004) solves this problem.

Load tests are a key issue in the deployment of web applications. JMeter (Halili, 2008) automates this testing, but has not been used in this work.

Finally, there are much more software security patterns than those used in this work. To include them in a multitier architecture is challenging task, but due to effort required for its implementation, it deserves another degree work.

ÍNDICE DE FIGURAS

● Capítulo 2.1

● Capítulo 2.2

- **Capítulo 2.3**

- **Capítulo 3.1**

- **Capítulo 4.1**

- **Capítulo 4.2**

- **Capítulo 4.3**

- **Capítulo 5.1**

REFERENCIAS Y BIBLIOGRAFÍA

Alur, D., Crupi, D., Malks, J. (2003). *Core J2EE Patterns: Best Practices and Design Strategies (2nd Edition)*. Prentice Hall, 2003.

Amuthan G (2014). *Spring MVC: Beginner's Guide*, 2014.

Balani, N., Hathi, R. (2009). *Apache CXF Web Service Development,* Packt Publishing, 2009.

Bartholomew, D. (2015). *Getting Started with MariaDB - Second Edition*, Packt Publishing, 2015.

Bell, P., Beer, B. (2014). *Introducing GitHub: A Non-Technical Guide*, O'REILLY, 2014.

Booch G. (1996). *Análisis y diseño orientado a objetos con aplicaciones, Segunda edición*, Addison-Wesley/Díaz de Santos, 1996

Brown, E. (2014). *Web Development with Node and Express: Leveraging the JavaScript Stack*, O'Reilly, 2014

Burke, B. (2013). *RESTful Java with JAX-RS 2.0*, O'Reilly 2013.

Cortés, H. & Navarro, A. (2017): *Enterprise WAE: A Lightweight UML Extension for the Characterization of the Presentation Tier of Enterprise Applications with MDD-Based Mockup Generation. International Journal of Software Engineering and Knowledge Engineering 27(8): 1291-1332 (2017)*.

Duckett, J. (2011). *HTML and CSS: Design and Build Websites*, 2011.

Duckett, J. (2014). *JavaScript and JQuery: Interactive Front-End Web Development*, Wiley, 2014.

Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley Professional, 1994.

Ferguson, J. (2011). *Jenkins: The Definitive Guide: Continuous Integration for the Masses*, O'REILLY, 2011.

Fowler, M. 2002. *Patterns of Enterprise Application Architecture. With David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, y Randy Stafford*. Addison-Wesley, 2002.

Freeman, A. (2017). *Pro ASP.NET Core MVC 2*, Apress, 2017.

Geary, D., Horstmann, C.S. (2010). *Core JavaServer Faces. 3rd Edition*. Prentice-Hall, 2010.

Halili, E. (2008). *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites*, Packt Publishing, 2008.

Hallam-Baker, P., Kaler, C., Monzillo, R., Nadalin, A. (2003). *Web Services Security: SOAP Message Security*, OASIS, 2003.

Hansen, M.D. (2007). *SOA Using Java Web Services*. Prentice Hall, 2007.

Hutten, D. (2017). *Git: Learn Version Control with Git: A step-by-step Ultimate beginners Guide* (English Edition), CreateSpace Independent Publishing Platform, 2017.

JSF & Spring MVC, 2018. https://www.dailyrazor.com/blog/jsf-vs-spring-mvc/

Jacobson, I. (2000). *El Proceso Unificado de Desarrollo de Software*, Addison Wesley Publishing Company, 2000.

Jim Conallen (2002): *Building Web Applications with UML (The Addison-Wesley Object-Technology Series)*, 2nd Edition, Addison Wesley, 2002.

Junit (2018). https://junit.org/junit5/

Keith, M., Schincariol, M., Nardone, M. (2018). *Pro JPA 2 in Java Ee 8: An In-Depth Guide to Java Persistence APIs*, Apress, 2018.

Little, M., Maron, J., Pavlik, G. (2004). *Java Transaction Processing: Design and Implementation*, Prentice Hall, 2004.

McLaughlin, B. (2002). *Java & XML Data Binding*, O'Reilly, 2002.

Mclaughlin, M. (2011). *Oracle Database 11g & MySQL 5.6 Developer Handbook (Oracle Press)*, McGraw-Hill Education, 2011.

Membrey, P. , Hows, D., Plugge, E. (2014). *MongoDB Basics*, Apress, 2014.

Mihalcea, V (2016). *High-Performance Java Persistence*, Vlad Mihalcea, 2016.

Mikowski, M. (2013). *Single Page Web Applications: JavaScript end-to-end*, Manning Publications, 2013.

Miles, R. (2006). *Learning UML 2.0*, O'Reilly, 2006.

Musciano, C (2006). *HTML & XHTML: The Definitive Guide*, O'REILLY, 2006.

Oaks, S. (2001). *Java Security (2nd Edition)*, O'Reilly, 2001.

Oracle JavaServer Faces Specification (2018): https://javaee.github.io/javaserverfaces-spec/ (último acceso septiembre 2018)

PrimeTek (2017). https://www.primefaces.org/

Schildt, H. (2014). *Java 8*, Anaya, 2014.

Schwaber, K., Sutherland, J. (2017). *The Scrum Guide*, 2017.

Scott, E. (2015). *SPA Design and Architecture: Understanding Single Page Web Applications*, Manning Publications, 2015.

Steel, C., Nagappan, R., Lai, R. (2005). *Core Security Patterns: Best Practices and Strategies for J2EE*, Web Services, and Identity Management, 2005.

Tansley, D. (1999). *Programación de shell Linux y Unix*, Addison-Wesley Professional, 1999.

Varanasi, B., Belida, S. (2014). *Introducing Maven 1st ed. Edition*, Apress, 2014.

Vukotic, A., Goodwill, J. (2011). *Apache Tomcat 7*, Springer Verlag GmbH, 2011.

Zambon, G. (2012). *Beginning JSP, JSF and Tomcat: Java Web Development*, Apress, 2012.